Space-Efficient Manifest Contracts

Michael Greenberg
Princeton University
mg19@cs.princeton.edu

Abstract

The standard algorithm for higher-order contract checking can lead to unbounded space consumption and can destroy tail recursion, altering a program's asymptotic space complexity. While space efficiency for gradual types—contracts mediating untyped and typed code—is well studied, sound space efficiency for manifest contracts—contracts that check stronger properties than simple types, e.g., "is a natural" instead of "is an integer"—remains an open problem.

We show how to achieve sound space efficiency for manifest contracts with strong predicate contracts. The essential trick is breaking the contract checking down into *coercions*: structured, blame-annotated lists of checks. By carefully preventing duplicate coercions from appearing, we can restore space efficiency while keeping the same observable behavior.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

Keywords contracts; pre- and post-conditions; function proxy; coercions; space efficiency

1. Introduction

Types are an extremely successful form of lightweight specification: programmers can state their intent—e.g., plus is a function that takes two numbers and returns another number—and then type checkers can ensure that a program conforms to the programmer's intent. Types can only go so far though: division is, like addition, a function that takes two numbers and returns another number... so long as the second number isn't zero. Conventional type systems do a good job of stopping many kinds of errors, but most type systems cannot protect partial operations like division and array indexing. Advanced techniques—singleton and dependent types, for example—can cover many of these cases, allowing programmers to use types like "non-zero number" or "index within bounds" to specify the domains on which partial operations are safe. Such techniques are demanding: they can be difficult to understand, they force certain programming idioms, and they place heavy constraints on the programming language, requiring purity or even strong normalization.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676967

Contracts are a popular compromise: programmers write type-like contracts of the form $\mathsf{Int} \rightarrow \{x{:}\mathsf{Int} \mid x \neq 0\} \rightarrow \mathsf{Int}$, where the predicates $x \neq 0$ are written in code. These type-like specifications can then be checked at runtime [5]. Models of contract calculi have taken two forms: latent and manifest [12]. We take the manifest approach here, which means checking contracts with *casts*, written $\langle T_1 \Rightarrow T_2 \rangle^l\ e$. Checking a *predicate contract* (also called a *refinement type*, though that term is overloaded) like $\{x{:}\mathsf{Int} \mid x \neq 0\}$ on a number n involves running the predicate $x \neq 0$ with n for x. Casts from one predicate contract to another, $\langle \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\} \rangle^l$, take a constant k and check to see that $e_2[k/x] \longrightarrow^* \mathsf{true}$. It's hard to know what to do with function casts at runtime: in $\langle T_{11} \rightarrow T_{12} \Rightarrow T_{21} \rightarrow T_{22} \rangle^l\ e$, we know that e is a $T_{11} \rightarrow T_{12}$, but what does that tell us about treating e as a $T_{21} \rightarrow T_{22}$? Findler and Felleisen's insight is that we must defer checking, waiting until the cast value e gets an argument [5]. These deferred checks are recorded on the value by means of a *function proxy*, i.e., $\langle T_{11} \rightarrow T_{12} \Rightarrow T_{21} \rightarrow T_{22} \rangle^l\ e$ is a value when e is a value; applying a function proxy unwraps it contravariantly. We check the domain contract T_1 on e, run the original function f on the result, and then check that result against the codomain contract T_2:

$$\begin{array}{l}(\langle T_{11} \rightarrow T_{12} \Rightarrow T_{21} \rightarrow T_{22} \rangle^l\ e_1)\ e_2 \longrightarrow \\ \langle T_{12} \Rightarrow T_{22} \rangle^l\ (e_1\ (\langle T_{21} \Rightarrow T_{11} \rangle^l\ e_2))\end{array}$$

Findler and Felleisen neatly designed a system for contract checking in a higher-order world, but there is a problem: contract checking is space inefficient [15].

Contract checking's space inefficiency can be summed up as follows: **function proxies break tail calls**. Calls to an unproxied function from a tail position can be optimized to not allocate stack frames. Proxied functions, however, will unwrap to have codomain contracts—breaking tail calls. We discuss other sources of space inefficiency below, but breaking tail calls is the most severe. Consider factorial written in accumulator passing style. The developer may believe that the following can be compiled to use tail calls:

$$\begin{array}{l}\mathsf{fact} : \{x{:}\mathsf{Int} \mid x \geq 0\} \rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\} \rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\} \\ \quad = \lambda x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\}.\ \lambda y{:}\{y{:}\mathsf{Int} \mid \mathsf{true}\}. \\ \qquad \mathsf{if}\ x = 0\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{fact}\ (x - 1)\ (x * y)\end{array}$$

A cast insertion algorithm [29] might produce the following non-tail recursive function:

$$\begin{array}{l}\mathsf{fact} = \\ \quad \langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \rightarrow \{y{:}\mathsf{Int} \mid \mathsf{true}\} \rightarrow \{z{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \\ \quad\ \{x{:}\mathsf{Int} \mid x \geq 0\} \rightarrow \{y{:}\mathsf{Int} \mid y \geq 0\} \rightarrow \{z{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_{\mathsf{fact}}} \\ \quad \lambda x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\}.\ \lambda y{:}\{y{:}\mathsf{Int} \mid \mathsf{true}\}. \\ \quad\ \mathsf{if}\ x = 0\ \mathsf{then}\ y\ \mathsf{else} \\ \qquad (\langle \{x{:}\mathsf{Int} \mid x \geq 0\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{true}\} \rangle^{l_{\mathsf{fact}}}\ (\mathsf{fact}\ \ldots))\end{array}$$

Tail-call optimization is essential for usable functional languages. Space inefficiency has been one of two significant obstacles for pervasive use of higher-order contract checking. (The other is state, which we do not treat here.)

In this work, we show how to achieve semantics-preserving space efficiency for non-dependent contract checking. Our approach is inspired by work on *gradual typing* [27], a form of (manifest) contracts designed to mediate dynamic and simple typing—that is, gradual typing (a) allows the dynamic type, and (b) restricts the predicates in contracts to checks on type tags. Herman et al. [15] developed the first space-efficient gradually typed system, using Henglein's coercions [14]; Siek and Wadler [28] devised a related system supporting blame. The essence of the solution is to allow casts to merge: given two adjacent casts $\langle T_2 \Rightarrow T_3 \rangle^{l_2}$ $(\langle T_1 \Rightarrow T_2 \rangle^{l_1}\ e)$, we must somehow combine them into a single cast. Siek and Wadler annotate their casts with an intermediate type representing the greatest lower bound of the types encountered. Such a trick doesn't work in our more general setting: simple types plus dynamic form a straightforward lattice using type precision as the ordering, but it's less clear what to do when we have arbitrary predicate contracts.

We define two modes of a single calculus, λ_{H}. The *classic* mode is just the conventional, inefficient semantics; the *eidetic* mode annotates casts with *refinement lists* and *function coercions*—a new form of coercion inspired by Greenberg [9]. The coercions keep track of checking so well that the type indices and blame labels on casts are unnecessary:

$$\langle T_2 \overset{c_2}{\Rightarrow} T_3 \rangle^{\bullet}\ (\langle T_1 \overset{c_1}{\Rightarrow} T_2 \rangle^{\bullet}\ e) \longrightarrow_{\mathsf{E}} \langle T_1 \overset{\mathsf{join}(c_1, c_2)}{\Rightarrow} T_3 \rangle^{\bullet}\ e$$

These coercions form a skew lattice: refinement lists have ordering constraints that break commutativity. Eidetic λ_{H} is space efficient and observationally equivalent to the classic mode.

Eidetic λ_{H} is the first manifest contract calculus that is both *sound* and *space efficient* with respect to the classic semantics—a result contrary to Greenberg [9], who conjectured that such a result is impossible. We believe that space efficiency is a critical step towards the implementation of practical languages with manifest contracts.

We do not prove a blame theorem [30], since we lack the clear separation of dynamic and static typing found in gradual typing. We conjecture that such a theorem could be proved. Our model has two limits worth mentioning: we do not handle dependency, a common and powerful feature in manifest systems; and, our bounds for space efficiency are *galactic*—they establish that contracts consume constant space, but do nothing to reduce that constant [20]. Our contribution is showing that sound space efficiency is *possible* where it was believed to be impossible [9]; we leave evidence that it is *practicable* for future work.

Our proofs are available in the extended version [10], Appendices A–C.

Readers who are very familiar with this topic can read Figures 1, 2, and 3 and then skip directly to Section 4. Readers who understand the space inefficiency of contracts but aren't particularly familiar with manifest contracts can skip Section 2 and proceed to Section 3.

2. Function proxies

Space inefficient contract checking breaks tail recursion—a show-stopping problem for realistic implementations of pervasive contract use. Racket's contract system [22], the most widely used higher-order contract system, takes a "macro" approach to contracts: contracts typically appear only on module interfaces, and aren't checked within a module. Their approach comes partly out of a philosophy of breaking invariants inside modules but not out of them, but also partly out of a need to retain tail recursion within modules. Space inefficiency has shaped the way their contract system has developed. They do not use our "micro" approach, wherein annotations and casts permeate the code.

Tail recursion aside, there is another important source of space inefficiency: the unbounded number of function proxies. Hierarchies of libraries are a typical example: consider a list library and a set library built using increasingly sorted lists. We might have:

$$\begin{array}{rcll} \mathsf{null} & : & \alpha\ \mathsf{List} \rightarrow \{x{:}\mathsf{Bool} \mid \mathsf{true}\} & = \ldots \\ \mathsf{head} & : & \{x{:}\alpha\ \mathsf{List} \mid \mathsf{not}\ (\mathsf{null}\ x)\} \rightarrow \alpha & = \ldots \\ \\ \mathsf{empty} & : & \alpha\ \mathsf{Set} \rightarrow \{x{:}\mathsf{Bool} \mid \mathsf{true}\} & = \mathsf{null} \\ \mathsf{min} & : & \{x{:}\alpha\ \mathsf{Set} \mid \mathsf{not}\ (\mathsf{empty}\ x)\} \rightarrow \alpha & = \mathsf{head} \end{array}$$

Our code reuse comes with a price: even though the precondition on min is effectively the same as that on head, we must have two function proxies, and the non-emptiness of the list representing the set is checked twice: first by checking empty, and again by checking null (which is the same function). Blame systems like those in Racket encourage modules to declare contracts to avoid being blamed, which can result in redundant checking like the above when libraries requirements imply sub-libraries' requirements.

Or consider a library of drawing primitives based around painters, functions of type Canvas→Canvas. An underlying graphics library offers basic functions for manipulating canvases and functions over canvases, e.g., primFlipH is a painter transformer—of type (Canvas→Canvas)→(Canvas→Canvas)—that flips the generated images horizontally. A wrapper library may add derived functions while re-exporting the underlying functions with refinement types specifying a canvas's square dimensions, where $\mathsf{SquareCanvas} = \{x{:}\mathsf{Canvas} \mid \mathsf{width}(x) = \mathsf{height}(x)\}$:

$$\begin{array}{l} \mathsf{flipH}\ p = \langle \mathsf{Canvas} \rightarrow \mathsf{Canvas} \Rightarrow \\ \qquad\qquad \mathsf{SquareCanvas} \rightarrow \mathsf{SquareCanvas} \rangle^{l} \\ \qquad\qquad\quad (\mathsf{primFlipH} \\ \qquad\qquad\qquad (\langle \mathsf{SquareCanvas} \rightarrow \mathsf{SquareCanvas} \Rightarrow \\ \qquad\qquad\qquad\quad \mathsf{Canvas} \rightarrow \mathsf{Canvas} \rangle^{l}\ p)) \end{array}$$

The wrapper library only accepts painters with appropriately refined types, but must strip away these refinements before calling the underlying implementation—which demands Canvas→Canvas painters. The wrapper library then has to cast these modified functions *back* to the refined types. Calling flipH (flipH p) yields:

$$\begin{array}{l} \langle \mathsf{Canvas} \rightarrow \mathsf{Canvas} \Rightarrow \mathsf{SquareCanvas} \rightarrow \mathsf{SquareCanvas} \rangle^{l} \\ \quad (\mathsf{primFlipH} \\ \quad\quad (\langle \mathsf{SquareCanvas} \rightarrow \mathsf{SquareCanvas} \Rightarrow \mathsf{Canvas} \rightarrow \mathsf{Canvas} \rangle^{l} \\ \quad\quad\quad (\langle \mathsf{Canvas} \rightarrow \mathsf{Canvas} \Rightarrow \mathsf{SquareCanvas} \rightarrow \mathsf{SquareCanvas} \rangle^{l} \\ \quad\quad\quad\quad (\mathsf{primFlipH} \\ \quad\quad\quad\quad\quad (\langle \mathsf{SquareCanvas} \rightarrow \mathsf{SquareCanvas} \Rightarrow \\ \quad\quad\quad\quad\quad\quad \mathsf{Canvas} \rightarrow \mathsf{Canvas} \rangle^{l}\ p))))) \end{array}$$

That is, we first cast p to a plain painter and return a new painter p'. We then cast p' into and then immediately out of the refined type, before continuing on to flip p'. All the while, we are accumulating many function proxies beyond the wrapping done by the underlying implementation of primFlipH. Redundant wrapping can become quite extreme, especially for continuation-passing programs. Function proxies are the essential problem: nothing bounds their accumulation. Unfolding unboundedly many function proxies creates stacks of unboundedly many checks—which breaks tail calls. A space-efficient scheme for manifest contracts bounds the number of function proxies that can accumulate.

3. Classic manifest contracts

The standard manifest contract calculus, λ_{H}, is originally due to Flanagan [7]. We give the syntax for the non-dependent fragment in Figure 1. We have highlighted in yellow the four syntactic forms relevant to contract checking. This paper paper discusses two modes of λ_{H}: classic λ_{H}, mode C,and eidetic λ_{H}, mode E. Each of these languages uses the syntax of Figure 1, while the typing rules

Modes

m ::= C classic λ_H; Section 3
 | E eidetic λ_H; Section 4

Types

$B ::= \mathsf{Bool} \mid \ldots$

$T ::= \{x{:}B \mid e\} \mid T_1 \to T_2$

Terms

$e ::= x \mid k \mid \lambda x{:}T.\ e \mid e_1\ e_2 \mid op(e_1, \ldots, e_n) \mid \langle T_1 \overset{a}{\Rightarrow} T_2 \rangle^l\ e \mid \langle \{x{:}B \mid e_1\}, e_2, k \rangle^l \mid \Uparrow l \mid \langle \{x{:}B \mid e_1\}, s, r, k, e \rangle^{\bullet}$

Annotations: type set, coercions, and refinement lists

$a ::= \bullet \mid c$

$c ::= r \mid c_1 \mapsto c_2$

$r ::= \mathsf{nil} \mid \{x{:}B \mid e\}^l, r$

Statuses

$s ::= \checkmark \mid ?$

Locations

$l ::= \bullet \mid l_1 \mid \ldots$

Figure 1. Syntax of λ_H

and operational semantics are indexed by the mode m. The proofs and metatheory are also mode-indexed. In an extended version of this work, we develop two additional modes with slightly different properties from eidetic λ_H, filling out a "framework" for space-efficient manifest contracts [10]. We omit the other two modes here to save space for eidetic λ_H, which is the only mode that is sound with respect to classic λ_H.

The metavariable B is used for base types, of which at least Bool must be present. There are two kinds of types. First, *predicate contracts* $\{x{:}B \mid e\}$, also called *refinements of base types* or just *refinement types*, denotes constants k of base type B such that $e[k/x]$ holds—that is, such that $e[k/x] \longrightarrow_m^* \mathsf{true}$ for any mode m. Function types $T_1 \to T_2$ are standard.

The terms of λ_H are largely those of the simply-typed lambda calculus: variables, constants k, abstractions, applications, and operations should all be familiar. The first distinguishing feature of λ_H's terms is the *cast*, written $\langle T_1 \overset{a}{\Rightarrow} T_2 \rangle^l\ e$. Here e is a term of type T_1; the cast checks whether e can be treated as a T_2—if e doesn't cut it, the cast will use its label l to raise the uncatchable exception $\Uparrow l$, read "blame l". Our casts also have annotations a. Classic doesn't need annotations—we write $\bullet$ and say "none". Eidetic λ_H uses coercions c, based on coercions in Henglein [14]. We explain coercions in greater detail in Section 4, but they amount to lists of blame-annotated refinement types r and function coercions.

The three remaining forms—active checks, blame, and coercion stacks—only occur as the program evaluates. Casts between refinement types are checked by *active checks* $\langle \{x{:}B \mid e_1\}, e_2, k \rangle^l$. The first term is the type being checked—necessary for the typing rule. The second term is the current status of the check; it is an invariant that $e_1[k/x] \longrightarrow_m^* e_2$. The final term is the constant being checked, which is returned wholesale if the check succeeds. When checks fail, the program raises *blame*, an uncatchable exception written $\Uparrow l$. A *coercion stack* $\langle \{x{:}B \mid e_1\}, s, r, k, e \rangle^{\bullet}$ represents the state of checking a coercion; we only use it in eidetic λ_H, so we postpone discussing it until Section 4.

3.1 Core operational semantics

Our mode-indexed operational semantics for our manifest calculi comprise three relations: $\mathsf{val}_m\ e$ identifies terms that are values in mode m (or m-values), $\mathsf{result}_m\ e$ identifies m-results, and $e_1 \longrightarrow_m e_2$ is the small-step reduction relation for mode m. Figure 2 defines the core rules. The rules for classic λ_H ($m = \mathsf{C}$) are in salmon; the shared space-efficient rules are in periwinkle. To save space, we pass over standard rules.

The mode-agnostic value rules are straightforward: constants are always values (V_Const), as are lambdas (V_Abs). Each mode defines its own value rule for function proxies, V_Proxy$m$. The classic rule, V_ProxyC, says that a function proxy

$$\langle T_{11} \to T_{12} \overset{\bullet}{\Rightarrow} T_{21} \to T_{22} \rangle^l\ e$$

is a C-value when e is a C-value. That is, function proxies can wrap lambda abstractions and other function proxies alike. Eidetic λ_H only allows lambda abstractions to be proxied. All of the space-efficient calculi in the literature take our approach, where a function cast applied to a value *is* a value; some space inefficient ones do, too [5, 12, 13]. In other formulations of λ_H in the literature, function proxies are implemented by introducing a new lambda as a wrapper à la Findler and Felleisen's $\overline{wrap}$ operator [1, 5, 7, 27]. Such an η-expansion semantics is convenient, since then applications only ever reduce by β-reduction. But it wouldn't suit our purposes at all: space efficiency demands that we combine function proxies. We can also imagine a third, ungainly semantics that looks into closures rather than having explicit function proxies. Results don't depend on the mode: m-values are always m-results (R_Val); blame is always a result, too (R_Blame).

E_Beta applies lambda abstractions via substitution, using a call-by-value rule. Note that β reduction in mode m requires that the argument is an m-value. The reduction rule for operations (E_Op) defers to operations' denotations, $[\![op]\!]$; since these may be partial (e.g., division), we assign types to operations that guarantee totality (see Section 3.2). That is, partial operations are a potential source of stuckness, and the types assigned to operations must guarantee the absence of stuckness. Robin Milner famously stated that "well typed expressions don't go wrong" [21]; his programs could go wrong by (a) applying a boolean like a function or (b) conditioning on a function like a boolean. Systems with more base types can go wrong in more ways, some of which are hard to capture in standard type systems. Contracts allow us to bridge that gap. Letting operations get stuck is a philosophical stance—contracts expand the notion of "wrong".

E_Unwrap applies function proxies to values, contravariantly in the domain and covariantly in the codomain. We also split up each cast's annotation, using $\mathsf{dom}(a)$ and $\mathsf{cod}(a)$—each mode is discussed in its respective section. E_CheckNoneC turns a cast between refinement types into an active check with the same blame label. We discard the source type—we already know that k is a $\{x{:}B \mid e_1\}$—and substitute the scrutinee into the target type, $e_2[k/x]$, as the current state of checking. We must also hold onto the scrutinee, in case the check succeeds. We are careful to not apply this rule in eidietc λ_H, which must generate annotations before running checks. Active checks evaluate by the congruence rule E_CheckInner until one of three results adheres: the predicate returns true, so the whole active check returns the scrutinee (E_CheckOK); the predicate returns false, so the whole active check raises blame using the label on the chceck (E_CheckFail); or blame was raised during checking, and we propagate it via E_CheckRaise. Checks in eidetic λ_H use slightly different forms, described in Section 4.

The core semantics includes several other congruence rules: E_AppL, E_AppR, and E_OpInner. Since space bounds rely not only on limiting the number of function proxies but also on accumulation of casts on the stack, the core semantics doesn't include a cast congruence rule. The congruence rule for casts in classic λ_H, E_CastInnerC, allows for free use of congruence. In the space-efficient calculus, the use of congruence is instead limited by the rules E_CastInnerE and E_CastMergeE. Cast arguments only take congruent steps when they aren't casts themselves. A cast ap-

Values and results $\boxed{\mathsf{val}_m\ e}$ $\boxed{\mathsf{result}_m\ e}$

$$\frac{}{\mathsf{val}_m\ k}\ \textsc{V\_Const} \qquad \frac{}{\mathsf{val}_m\ \lambda x{:}T.\ e}\ \textsc{V\_Abs} \qquad \frac{\mathsf{val}_\mathsf{C}\ e}{\mathsf{val}_\mathsf{C}\ \langle T_{11}\to T_{12} \overset{\bullet}{\Rightarrow} T_{21}\to T_{22}\rangle^l\ e}\ \textsc{V\_ProxyC} \qquad \frac{\mathsf{val}_m\ e}{\mathsf{result}_m\ e}\ \textsc{R\_Val} \qquad \frac{}{\mathsf{result}_m\ \Uparrow l}\ \textsc{R\_Blame}$$

Shared operational semantics $\boxed{e_1 \longrightarrow_m e_2}$

$$\frac{\mathsf{val}_m\ e_2}{(\lambda x{:}T.\ e_{12})\ e_2 \longrightarrow_m e_{12}[e_2/x]}\ \textsc{E\_Beta} \qquad \frac{\mathsf{val}_m\ e_1\ \ldots\ \mathsf{val}_m\ e_n}{op(e_1,\ \ldots,\ e_n) \longrightarrow_m [\![op]\!]\,(e_1,\ \ldots,\ e_n)}\ \textsc{E\_Op}$$

$$\frac{\mathsf{val}_m\ \langle T_{11}\to T_{12} \overset{a}{\Rightarrow} T_{21}\to T_{22}\rangle^l\ e_1 \quad \mathsf{val}_m\ e_2}{(\langle T_{11}\to T_{12} \overset{a}{\Rightarrow} T_{21}\to T_{22}\rangle^l\ e_1)\ e_2 \longrightarrow_m \langle T_{12} \overset{\mathsf{cod}(a)}{\Rightarrow} T_{22}\rangle^l\ (e_1\ (\langle T_{21} \overset{\mathsf{dom}(a)}{\Rightarrow} T_{11}\rangle^l\ e_2))}\ \textsc{E\_Unwrap}$$

$$\begin{aligned} \mathsf{dom}(\bullet) &= \bullet & \qquad \mathsf{cod}(\bullet) &= \bullet \\ \mathsf{dom}(c_1 \mapsto c_2) &= c_1 & \qquad \mathsf{cod}(c_1 \mapsto c_2) &= c_2 \end{aligned}$$

$$\frac{}{\langle \{x{:}B \mid e_1\} \overset{\bullet}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^l\ k \longrightarrow_\mathsf{C} \langle \{x{:}B \mid e_2\}, e_2[k/x], k\rangle^l}\ \textsc{E\_CheckNoneC}$$

$$\frac{}{\langle \{x{:}B \mid e\}, \mathsf{true}, k\rangle^l \longrightarrow_m k}\ \textsc{E\_CheckOK} \qquad \frac{}{\langle \{x{:}B \mid e\}, \mathsf{false}, k\rangle^l \longrightarrow_m \Uparrow l}\ \textsc{E\_CheckFail}$$

$$\frac{e_1 \longrightarrow_m e_1'}{e_1\ e_2 \longrightarrow_m e_1'\ e_2}\ \textsc{E\_AppL} \qquad \frac{\mathsf{val}_m\ e_1 \quad e_2 \longrightarrow_m e_2'}{e_1\ e_2 \longrightarrow_m e_1\ e_2'}\ \textsc{E\_AppR}$$

$$\frac{\mathsf{val}_m\ e_1\ \ldots\ \mathsf{val}_m\ e_{i-1} \quad e_i \longrightarrow_m e_i'}{op(e_1, \ldots, e_{i-1}, e_i, \ldots, e_n) \longrightarrow_m op(e_1, \ldots, e_{i-1}, e_i', \ldots, e_n)}\ \textsc{E\_OpInner}$$

$$\frac{e \longrightarrow_\mathsf{C} e'}{\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e \longrightarrow_\mathsf{C} \langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e'}\ \textsc{E\_CastInnerC} \qquad \frac{e_2 \longrightarrow_m e_2'}{\langle \{x{:}B \mid e_1\}, e_2, k\rangle^l \longrightarrow_m \langle \{x{:}B \mid e_1\}, e_2', k\rangle^l}\ \textsc{E\_CheckInner}$$

$$\frac{e_2 \longrightarrow_\mathsf{E} e_2' \quad e_2 \neq \langle T_1 \overset{a'}{\Rightarrow} T_2\rangle^{l'}\ e_2''}{\langle T_2 \overset{a}{\Rightarrow} T_3\rangle^l\ e_2 \longrightarrow_\mathsf{E} \langle T_2 \overset{a}{\Rightarrow} T_3\rangle^l\ e_2'}\ \textsc{E\_CastInnerE} \qquad \frac{c_3 = \mathsf{join}(c_1, c_2)}{\langle T_2 \overset{c_2}{\Rightarrow} T_3\rangle^l\ (\langle T_1 \overset{c_1}{\Rightarrow} T_2\rangle^{l'}\ e_2) \longrightarrow_\mathsf{E} \langle T_1 \overset{c_3}{\Rightarrow} T_3\rangle^l\ e_2}\ \textsc{E\_CastMergeE}$$

$$\frac{}{\Uparrow l\ e_2 \longrightarrow_m \Uparrow l}\ \textsc{E\_AppRaiseL} \qquad \frac{\mathsf{val}_m\ e_1}{e_1\ \Uparrow l \longrightarrow_m \Uparrow l}\ \textsc{E\_AppRaiseR} \qquad \frac{}{\langle T_1 \overset{S}{\Rightarrow} T_2\rangle^l\ \Uparrow l' \longrightarrow_m \Uparrow l'}\ \textsc{E\_CastRaise}$$

$$\frac{\mathsf{val}_m\ e_1\ \ldots\ \mathsf{val}_m\ e_{i-1}}{op(e_1, \ldots, e_{i-1}, \Uparrow l, \ldots, e_n) \longrightarrow_m \Uparrow l}\ \textsc{E\_OpRaise} \qquad \frac{}{\langle \{x{:}B \mid e\}, \Uparrow l, k\rangle^{l'} \longrightarrow_m \Uparrow l}\ \textsc{E\_CheckRaise}$$

Figure 2. Core operational semantics of λ_H; classic λ_H rules are salmon; space-efficient rules are periwinkle

plied to another cast *merges*, using the join function. Each space-efficient calculus uses a different annotation scheme, so each one has a different merge function. We are careful to define join only over coercions, so E_CastMergeE won't apply on the empty annotation, $\bullet$ (read "none"). We have E_CastMergeE arbitrarily retain the label of the outer cast. This choice is ultimately irrelevant, since eidetic λ_H won't need to keep track of blame labels on casts themselves (Section 4). In addition to congruence rules, there are blame propagation rules, which are universal: E_AppRaiseL, E_AppRaiseR, E_CastRaise, E_OpRaise. These rules propagate the uncatchable exception $\Uparrow l$ while obeying call-by-value rules.

3.2 Type system

All modes share a type system, given in Figure 3. All judgments are universal and simply thread the mode through—except for annotation well formedness $\vdash_m a \parallel T_1 \Rightarrow T_2$, which is mode specific, and a single eidetic-specific rule given in Figure 4. The type system comprises several relations: context well formedness $\vdash_m \Gamma$ and type well formedness $\vdash_m T$; type compatibility $\vdash T_1 \parallel T_2$, a mode-less comparison of the *skeleton* of two types; annotation well formedness $\vdash_m a \parallel T_1 \Rightarrow T_2$; and term typing $\Gamma \vdash_m e : T$.

Context well formedness is entirely straightforward; type well formedness requires some care to get base types off the ground. We establish as an axiom that the *raw* type $\{x{:}B \mid \mathsf{true}\}$ is well formed for every base type B (WF_Base); we then use raw types to check that refinements are well formed: $\{x{:}B \mid e\}$ is well formed in mode m if e is well typed as a boolean in mode m when x is a value of type B (WF_Refine). Without WF_Base, WF_Refine wouldn't have a well formed context. Function types are well formed in mode m when their domains and codomains are well formed in mode m. (Unlike many recent formulations, our functions are not dependent—we leave dependency as future work.) Type compatibility $\vdash T_1 \parallel T_2$ identifies types which can be cast to each other: the types must have the same "skeleton". It is reasonable to try to cast a non-zero integer $\{x{:}\mathsf{Int} \mid x \neq 0\}$ to a positive integer $\{x{:}\mathsf{Int} \mid x > 0\}$, but it is senseless to cast it to a boolean $\{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ or to a function type $T_1 \to T_2$. Every cast must be between compatible types; at their core, λ_H programs are simply typed lambda calculus programs. Type compatibility

Context and type well formedness $\boxed{\vdash_m \Gamma}$ $\boxed{\vdash_m T}$

$$\frac{}{\vdash_m \emptyset}\ \text{WF\_EMPTY} \qquad \frac{\vdash_m \Gamma \quad \vdash_m T}{\vdash_m \Gamma, x{:}T}\ \text{WF\_EXTEND}$$

$$\frac{}{\vdash_m \{x{:}B \mid \mathsf{true}\}}\ \text{WF\_BASE} \qquad \frac{x{:}\{x{:}B \mid \mathsf{true}\} \vdash_m e : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\vdash_m \{x{:}B \mid e\}}\ \text{WF\_REFINE} \qquad \frac{\vdash_m T_1 \quad \vdash_m T_2}{\vdash_m T_1 \to T_2}\ \text{WF\_FUN}$$

Type compatibility and annotation well formedness $\boxed{\vdash T_1 \parallel T_2}$ $\boxed{\vdash_m a \parallel T_1 \Rightarrow T_2}$

$$\frac{}{\vdash \{x{:}B \mid e_1\} \parallel \{x{:}B \mid e_2\}}\ \text{S\_REFINE} \qquad \frac{\vdash T_{11} \parallel T_{21} \quad \vdash T_{12} \parallel T_{22}}{\vdash T_{11}\to T_{12} \parallel T_{21}\to T_{22}}\ \text{S\_FUN} \qquad \frac{\vdash T_1 \parallel T_2 \quad \vdash_m T_1 \quad \vdash_m T_2}{\vdash_m \bullet \parallel T_1 \Rightarrow T_2}\ \text{A\_NONE}$$

Expression typing $\boxed{\Gamma \vdash_m e : T}$

$$\frac{\vdash_m \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash_m x : T}\ \text{T\_VAR} \qquad \frac{\vdash_m T_1 \quad \Gamma, x{:}T_1 \vdash_m e_{12} : T_2}{\Gamma \vdash_m \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \text{T\_ABS} \qquad \frac{\vdash_m \Gamma \quad \vdash_m T}{\Gamma \vdash_m \Uparrow l : T}\ \text{T\_BLAME}$$

$$\frac{\vdash_m \Gamma \quad \vdash_m \{x{:}B \mid e\} \quad \mathsf{ty}(k) = B \quad e[k/x] \longrightarrow_m^* \mathsf{true}}{\Gamma \vdash_m k : \{x{:}B \mid e\}}\ \text{T\_CONST} \qquad \frac{\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T \quad \Gamma \vdash_m e_i : T_i}{\Gamma \vdash_m op(e_1, \ldots, e_n) : T}\ \text{T\_OP}$$

$$\frac{\Gamma \vdash_m e_1 : (T_1 \to T_2) \quad \Gamma \vdash_m e_2 : T_1}{\Gamma \vdash_m e_1\ e_2 : T_2}\ \text{T\_APP} \qquad \frac{\vdash_m a \parallel T_1 \Rightarrow T_2 \quad \Gamma \vdash_m e : T_1}{\Gamma \vdash_m \langle T_1 \overset{a}{\Rightarrow} T_2 \rangle^l\ e : T_2}\ \text{T\_CAST}$$

$$\frac{\vdash_m \Gamma \quad \vdash_m \{x{:}B \mid e_1\} \quad \mathsf{ty}(k) = B \quad \emptyset \vdash_m e_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \quad e_1[k/x] \longrightarrow_m^* e_2}{\Gamma \vdash_m \langle \{x{:}B \mid e_1\}, e_2, k \rangle^l : \{x{:}B \mid e_1\}}\ \text{T\_CHECK}$$

Figure 3. Universal typing rules of λ_{H}

is reflexive, symmetric, and transitive; i.e., it is an equivalence relation.

Our family of calculi use different annotations. All source programs (defined below) begin without annotations—we write the empty annotation $\bullet$, read "none". The universal annotation well formedness rule just defers to type compatibility (A_NONE); it is an invariant that $\vdash_m a \parallel T_1 \Rightarrow T_2$ implies $\vdash T_1 \parallel T_2$.

As for term typing, the T_VAR, T_ABS, T_OP, and T_APP rules are entirely conventional. T_BLAME types blame at any (well formed) type. A constant k can be typed by T_CONST at any type $\{x{:}B \mid e\}$ in mode m if: (a) k is a B, i.e., $\mathsf{ty}(k) = B$; (b) the type in question is well formed in m; and (c), if $e[k/x] \longrightarrow_m^* \mathsf{true}$. As an immediate consequence, we can derive the following rule typing constants at their raw type, since $\mathsf{true} \longrightarrow_m^* \mathsf{true}$ in all modes and raw types are well formed in all modes (WF_BASE):

$$\frac{\vdash_m \Gamma \qquad \mathsf{ty}(k) = B}{\Gamma \vdash_m k : \{x{:}B \mid \mathsf{true}\}}$$

This approach to typing constants in a manifest calculus is novel: it offers a great deal of latitude with typing, while avoiding the subtyping of some formulations [7, 12, 17, 18] and the extra rule of others [1]. We assume that $\mathsf{ty}(k) = \mathsf{Bool}$ iff $k \in \{\mathsf{true}, \mathsf{false}\}$.

We require in T_OP that $\mathsf{ty}(op)$ only produces well formed first-order types, i.e., types of the form $\vdash_m \{x{:}B_1 \mid e_1\} \to \ldots \to \{x{:}B_n \mid e_n\}$. We require that the type is consistent with the operation's denotation: $[\![op]\!]\,(k_1, \ldots, k_n)$ is defined iff $e_i[k_i/x] \longrightarrow_m^* \mathsf{true}$ for all m. For this evaluation to hold for every system we consider, the types assigned to operations can't involve casts that both (a) stack and (b) can fail. We believe this is not so stringent a requirement: the types for operations ought to be simple, e.g. $\mathsf{ty}(\mathsf{div}) = \{x{:}\mathsf{Real} \mid \mathsf{true}\} \to \{y{:}\mathsf{Real} \mid y \neq 0\} \to \{z{:}\mathsf{Real} \mid \mathsf{true}\}$, and stacked casts only arise in stack-free terms due to function proxies. In general, it is interesting to ask what refinement types to assign to constants, as careless assignments can lead to circular checking (e.g., if division has a codomain cast checking its work with multiplication and vice versa).

The typing rule for casts, T_CAST, relies on the annotation well formedness rule: $\langle T_1 \overset{a}{\Rightarrow} T_2 \rangle^l\ e$ is well formed in mode m when $\vdash_m a \parallel T_1 \Rightarrow T_2$ and e is a T_1. Allowing any cast between compatible base types is conservative: a cast from $\{x{:}\mathsf{Int} \mid x > 0\}$ to $\{x{:}\mathsf{Int} \mid x \leq 0\}$ always fails. Earlier work has used SMT solvers to try to statically reject certain casts and eliminate those that are guaranteed to succeed [2, 7, 18]; we omit these checks, as we view them as secondary—a static analysis offering bug-finding and optimization, and not the essence of the system.

The final rule, T_CHECK, is used for checking active checks, which should only occur at runtime. In fact, they should only ever be applied to closed terms; the rule allows for any well formed context as a technical device for weakening.

Active checks $\langle \{x{:}B \mid e_1\}, e_2, k \rangle^l$ arise as the result of casts between refined base types, as in the following classic λ_{H} evaluation of a successful cast:

$$\begin{aligned} \langle \{x{:}B \mid e\} \overset{\bullet}{\Rightarrow} \{x{:}B \mid e'\} \rangle^l\ k &\longrightarrow_{\mathsf{C}} \langle \{x{:}B \mid e'\}, e'[k/x], k \rangle^l \\ &\longrightarrow_{\mathsf{C}}^* \langle \{x{:}B \mid e'\}, \mathsf{true}, k \rangle^l \\ &\longrightarrow_{\mathsf{C}} k \end{aligned}$$

If we are going to prove type soundness via syntactic methods [32], we must have enough information to type k at $\{x{:}B \mid e'\}$. For this reason, T_CHECK requires that $e_1[k/x] \longrightarrow_m^* e_2$; this way, we know that $e'[k/x] \longrightarrow_m^* \mathsf{true}$ at the end of the previous derivation, which is enough to apply T_CONST. The other premises of T_CHECK ensure that the types all match up: that the target refinement type is well formed; that k has the base type in question; and that e_2, the current state of the active check, is also well formed.

To truly say that our languages share a syntax and a type system, we highlight a subset of type derivations as *source program* type derivations. We show that source programs well typed in one mode are well typed in the all modes [10].

3.1 Definition [Source program]: A source program type derivation obeys the following rules:

– T_CONST only ever assigns the type $\{x{:}\mathsf{ty}(k) \mid \mathsf{true}\}$. Variations in each mode's evaluation aren't reflected in the (source program) type system. (We could soundly relax this requirement to allow $\{x{:}\mathsf{ty}(k) \mid e\}$ such that $e[k/x] \longrightarrow_m^* \mathsf{true}$ for any mode m.)

– Casts have empty annotations $a = \bullet$. Casts also have blame labels, and not empty blame (also written $\bullet$).

– T_CHECK, T_STACK (Section 4), and T_BLAME are not used—these are for runtime only.

Note that source programs don't use any of the typing rules that defer to the evaluation relation (T_CHECK and T_STACK), so we can maintain a clear phase distinction between type checking programs and running them.

3.3 Metatheory

One distinct advantage of having a single syntax with parameterized semantics is that some of the metatheory can be done once for all modes. Each mode proves its own canonical forms lemma—since each mode has a unique notion of value—and its own progress and preservation lemmas for syntactic type soundness [32]. But other standard metatheoretical machinery—weakening, substitution, and regularity—can be proved for all modes at once (see Section A.1). To wit, we prove syntactic type soundness in Appendix A.2 for classic λ_{H} in just three mode-specific lemmas: canonical forms, progress, and preservation. In every theorem statement, we include a reference to the lemma number where it is proved in the appendix. In PDF versions, this reference is hyperlinked.

Lemma [Classic canonical forms (A.11)]: If $\emptyset \vdash_{\mathsf{C}} e : T$ and $\mathsf{val}_{\mathsf{C}}\ e$ then:

– If $T = \{x{:}B \mid e'\}$, then $e = k$ and $\mathsf{ty}(k) = B$ and $e'[e/x] \longrightarrow_{\mathsf{C}}^* \mathsf{true}$.
– If $T = T_1 \to T_2$, then either $e = \lambda x{:}T.\ e'$ or $e = \langle T_{11} \to T_{12} \overset{\bullet}{\Rightarrow} T_{21} \to T_{22} \rangle^l\ e'$.

Lemma [Classic progress (A.12)]: If $\emptyset \vdash_{\mathsf{C}} e : T$, then either:

1. $\mathsf{result}_{\mathsf{C}}\ e$, i.e., $e = \Uparrow l$ or $\mathsf{val}_{\mathsf{C}}\ e$; or
2. there exists an e' such that $e \longrightarrow_{\mathsf{C}} e'$.

Lemma [Classic preservation (A.13)]: If $\emptyset \vdash_{\mathsf{C}} e : T$ and $e \longrightarrow_{\mathsf{C}} e'$, then $\emptyset \vdash_{\mathsf{C}} e' : T$.

4. Eidetic space efficiency

Eidetic λ_{H} uses *coercions*. Coercions do two critical things: they retain check order, and they track blame. Our coercions are ultimately inspired by those of Henglein [14]; we discuss the relationship between our coercions and his in related work (Section 7). Recall the syntax of coercions from Figure 1:

$$c ::= r \mid c_1 \mapsto c_2$$
$$r ::= \mathsf{nil} \mid \{x{:}B \mid e\}^l, r$$

Coercions come in two flavors: blame-annotated refinement lists r—zero or more refinement types, each annotated with a blame label—and function coercions $c_1 \mapsto c_2$. We write them as comma separated lists, omitting the empty refinement list nil when the refinement list is non-empty. We define the coercion well formedness rules, an additional typing rule, and reduction rules for eidetic λ_{H} in Figure 4. To ease the exposition, our explanation doesn't mirror the rule groupings in the figure.

As a general intuition, coercions are plans for checking: they contain precisely those types to be checked. Refinement lists are well formed for casts between $\{x{:}B \mid e_1\}$ and $\{x{:}B \mid e_2\}$ when: (a) every type in the list is a blame-annotated, well formed refinement of B, i.e., all the types are of the form $\{x{:}B \mid e\}^l$ and are therefore similar to the indices; (b) there are no duplicated types in the list; and (c) the target type $\{x{:}B \mid e_2\}$ is implied by some other type in the list. Note that the input type for all refinement lists can be any well formed refinement—this corresponds to the intuition that base types have no negative parts, i.e., casts between refinements ignore the type on the left. Finally, we simply write "no duplicates in r"—it is an invariant during the evaluation of source programs. Function coercions, on the other hand, have a straightforward (contravariant) well formedness rule.

The E_COERCE rule translates source-program casts to coercions: $\mathsf{coerce}(T_1, T_2, l)$ is a coercion representing exactly the checking done by the cast $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l$. All of the refinement types in $\mathsf{coerce}(T_1, T_2, l)$ are annotated with the blame label l, since that's the label that would be blamed if the cast failed at that type. Since a coercion is a complete plan for checking, a coercion annotation obviates the need for type indices and blame labels. To this end, E_COERCE drops the blame label from the cast, replacing it with an empty label. We keep the type indices so that we can reuse E_CASTMERGEE from the universal semantics, and also as a technical device in the preservation proof.

The actual checking of coercions rests on the treatment of refinement lists: function coercions are expanded as functions are applied by E_UNWRAP, so they don't need much special treatment beyond a definition for dom and cod. Eidetic λ_{H} uses *coercion stacks* $\langle \{x{:}B \mid e_1\}, s, r, k, e_2 \rangle^\bullet$ to evaluate refinement lists. Coercion stacks are type checked by T_STACK (in Figure 4). We explain the operational semantics before explaining the typing rule. Coercion stacks are runtime-only entities comprising five parts: a target type, a status, a pending refinement list, a constant scrutinee, and a checking term. We keep the target type of the coercion for preservation's sake. The status bit s is either $\checkmark$ or ?: when the status is $\checkmark$, we are currently checking or have already checked the target type $\{x{:}B \mid e_1\}$; when it is ?, we haven't. The pending refinement list r holds those checks not yet done. When $s = ?$, the target type is still in r. The scrutinee k is the constant we're checking; the checking term e_2 is *either* the scrutinee k itself, or it is an active check on k.

The evaluation of a coercion stack proceeds as follows. First, E_COERCESTACK starts a coercion stack when a cast between refinements meets a constant, recording the target type, setting the status to ?, and setting the checking term to k. Then E_STACKPOP starts an active check on the first type in the refinement list, using its blame label on the active check—possibly updating the status if the type being popped from the list is the target type. The active check runs by the congruence rule E_STACKINNER, eventually returning k itself or blame. In the latter case, E_STACKRAISE propagates the blame. If not, then the scrutinee is k once more and E_STACKPOP can fire again. Eventually, the refinement list is exhausted, and E_STACKDONE returns k.

Now we can explain T_STACK's many jobs. It must recapitulate A_REFINE, but not exactly—since eventually the target type will be checked and no longer appear in r. The status s differentiates what our requirement is: when $s = ?$, the target type is in r. When $s = \checkmark$, we either know that k inhabits the target type or that we are currently checking the target type (i.e., an active check of the target type at some blame label reduces to our current checking term).

Finally, we need to define how to merge casts. We use the join operator, which is very nearly concatenation on refinement lists and a contravariant homomorphism on function coercions. It's not

Coercion implication predicate: axioms $\boxed{\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}}$

(**Reflexivity**) If $\vdash_{\mathsf{E}} \{x{:}B \mid e\}$ then $\{x{:}B \mid e\} \supset \{x{:}B \mid e\}$.
(**Transitivity**) If $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}$ and $\{x{:}B \mid e_2\} \supset \{x{:}B \mid e_3\}$ then $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_3\}$.
(**Adequacy**) If $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}$ then $\forall k \in \mathcal{K}_B.\ e_1[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true}$ implies $e_2[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true}$.
(**Decidability**) For all $\vdash_{\mathsf{E}} \{x{:}B \mid e_1\}$ and $\vdash_{\mathsf{E}} \{x{:}B \mid e_2\}$, it is decidable whether $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}$.

Coercion well formedness and term typing $\boxed{\vdash_m c \parallel T_1 \Rightarrow T_2}$ $\boxed{\Gamma \vdash_m e : T}$

$$\frac{\begin{array}{l}\vdash_{\mathsf{E}} \{x{:}B \mid e_1\} \quad \vdash_{\mathsf{E}} \{x{:}B \mid e_2\} \\ \forall \{x{:}B \mid e\} \in r.\ \vdash_{\mathsf{E}} \{x{:}B \mid e\} \quad \text{no duplicates in } r \\ \exists \{x{:}B \mid e\} \in r.\ \{x{:}B \mid e\} \supset \{x{:}B \mid e_2\}\end{array}}{\vdash_{\mathsf{E}} r \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}} \ \text{A\_REFINE}$$

$$\frac{\vdash_{\mathsf{E}} c_1 \parallel T_{21} \Rightarrow T_{11} \quad \vdash_{\mathsf{E}} c_2 \parallel T_{12} \Rightarrow T_{22}}{\vdash_{\mathsf{E}} c_1 \mapsto c_2 \parallel (T_{11}\to T_{12}) \Rightarrow (T_{21}\to T_{22})} \ \text{A\_FUN}$$

$$\frac{\begin{array}{l}\vdash_{\mathsf{E}} \Gamma \quad \vdash_{\mathsf{E}} \{x{:}B \mid e_1\} \quad \mathsf{ty}(k) = B \quad \emptyset \vdash_{\mathsf{E}} e_2 : \{x{:}B \mid e_3\} \quad \forall \{x{:}B \mid e\} \in r.\ \vdash_{\mathsf{E}} \{x{:}B \mid e\} \\ s = \checkmark \text{ implies } e_1[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true} \vee (\exists \{x{:}B \mid e\}.\ \exists l.\ \{x{:}B \mid e\} \supset \{x{:}B \mid e_1\} \wedge \langle \{x{:}B \mid e\}, e[k/x], k\rangle^l \longrightarrow^*_{\mathsf{E}} e_2) \\ s = ?\text{ implies } (\exists \{x{:}B \mid e\} \in r.\ \{x{:}B \mid e\} \supset \{x{:}B \mid e_1\})\end{array}}{\Gamma \vdash_{\mathsf{E}} \langle \{x{:}B \mid e_1\}, s, r, k, e_2\rangle^\bullet : \{x{:}B \mid e_1\}} \ \text{T\_STACK}$$

Values and operational semantics $\boxed{\mathsf{val}_{\mathsf{E}}\ e}$ $\boxed{e_1 \longrightarrow_{\mathsf{E}} e_2}$

$$\frac{}{\mathsf{val}_{\mathsf{E}}\ \langle T_{11}\to T_{12} \overset{c_1\mapsto c_2}{\Rightarrow} T_{21}\to T_{22}\rangle^\bullet\ \lambda x{:}T.\ e} \ \text{V\_PROXYE}$$

$$\frac{}{\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e \longrightarrow_{\mathsf{E}} \langle T_1 \overset{\mathsf{coerce}(T_1,T_2,l)}{\Rightarrow} T_2\rangle^\bullet\ e} \ \text{E\_COERCE}$$

$$\frac{}{\langle \{x{:}B \mid e_1\} \overset{r}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^\bullet\ k \longrightarrow_{\mathsf{E}} \langle \{x{:}B \mid e_2\}, ?, r, k, k\rangle^\bullet} \ \text{E\_COERCESTACK}$$

$$\frac{}{\langle \{x{:}B \mid e\}, s, (\{x{:}B \mid e'\}^l, r), k, k\rangle^\bullet \longrightarrow_{\mathsf{E}} \langle \{x{:}B \mid e\}, s \vee (e = e'), r, k, \langle \{x{:}B \mid e'\}, e'[k/x], k\rangle^l\rangle^\bullet} \ \text{E\_STACKPOP}$$

$$\frac{e' \longrightarrow_{\mathsf{E}} e''}{\langle \{x{:}B \mid e\}, s, r, k, e'\rangle^\bullet \longrightarrow_{\mathsf{E}} \langle \{x{:}B \mid e\}, s, r, k, e''\rangle^\bullet} \ \text{E\_STACKINNER}$$

$$\frac{}{\langle \{x{:}B \mid e\}, s, r, k, \Uparrow l'\rangle^\bullet \longrightarrow_{\mathsf{E}} \Uparrow l'} \ \text{E\_STACKRAISE}$$

$$\frac{}{\langle \{x{:}B \mid e\}, \checkmark, \mathsf{nil}, k, k\rangle^\bullet \longrightarrow_{\mathsf{E}} k} \ \text{E\_STACKDONE}$$

Cast translation and coercion operations

$$\begin{array}{rcl} \mathsf{dom}(c_1 \mapsto c_2) &=& c_1 \\ \mathsf{cod}(c_1 \mapsto c_2) &=& c_2 \end{array} \qquad \begin{array}{rcl} \mathsf{coerce}(\{x{:}B \mid e_1\}, \{x{:}B \mid e_2\}, l) &=& \{x{:}B \mid e_2\}^l \\ \mathsf{coerce}(T_{11}\to T_{12}, T_{21}\to T_{22}, l) &=& \mathsf{coerce}(T_{21}, T_{11}, l) \mapsto \mathsf{coerce}(T_{12}, T_{22}, l) \end{array}$$

$$\begin{array}{rcl} \mathsf{join}(\{x{:}B \mid e\}^l, \mathsf{nil}) &=& \{x{:}B \mid e\}^l \\ \mathsf{join}(\{x{:}B \mid e\}^l, r) &=& \{x{:}B \mid e\}^l, \mathsf{drop}\,(r, \{x{:}B \mid e\}) \end{array} \qquad \begin{array}{rcl} \mathsf{join}(\mathsf{nil}, r_2) &=& r_2 \\ \mathsf{join}((\{x{:}B \mid e\}^l, r_1), r_2) &=& \mathsf{join}(\{x{:}B \mid e\}^l, \mathsf{join}(r_1, r_2)) \\ \mathsf{join}(c_{11} \mapsto c_{12}, c_{21} \mapsto c_{22}) &=& \mathsf{join}(c_{21}, c_{11}) \mapsto \mathsf{join}(c_{12}, c_{22}) \end{array}$$

$$\begin{array}{rcl} \mathsf{drop}\,(\mathsf{nil}, \{x{:}B \mid e\}) &=& \mathsf{nil} \\ \mathsf{drop}\,((\{x{:}B \mid e_1\}^l, r), \{x{:}B \mid e\}) &=& \begin{cases} \mathsf{drop}\,(r, \{x{:}B \mid e\}) & \{x{:}B \mid e\} \supset \{x{:}B \mid e_1\} \\ \{x{:}B \mid e_1\}^l, \mathsf{drop}\,(r, \{x{:}B \mid e\}) & \{x{:}B \mid e\} \not\supset \{x{:}B \mid e_1\} \end{cases} \end{array} \qquad \begin{array}{rcl} \checkmark \vee (e_1 = e_2) &=& \checkmark \\ ? \vee (e_1 = e_2) &=& \begin{cases} \checkmark & e_1 = e_2 \\ ? & \text{otherwise} \end{cases} \end{array}$$

Figure 4. Typing rules and operational semantics for eidetic λ_{H}

concatenation because it uses an implication predicate, the pre-order $\supset$, to eliminate duplicates (because $\supset$ is reflexive) and hide subsumed types (because $\supset$ is adequate). We read $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}$ as "$\{x{:}B \mid e_1\}$ implies $\{x{:}B \mid e_2\}$". When eliminating types, join always chooses the leftmost blame label. Contravariance means that $\mathsf{join}(c_1, c_2)$ takes leftmost labels in positive positions and rightmost labels in negative ones. The coerce metafunction and join operator work together to make sure that the refinement lists are correctly ordered. As we show below, 'correctly ordered' means the positive parts take older labels and negative parts take newer ones. E_CASTMERGEE is slightly subtle—we never merge casts with $\bullet$ as an annotation because such merges aren't defined.

In Figure 4, we only give the axioms for $\supset$: it must be an adequate, decidable pre-order. Syntactic type equality is the simplest implementation of the $\supset$ predicate, but the reflexive transitive closure of any adequate decidable relation would work.

By way of example, consider a cast from $T_1 = \{x{:}\mathsf{Int} \mid x \geq 0\}\to\{x{:}\mathsf{Int} \mid x \geq 0\}$ to $T_2 = \{x{:}\mathsf{Int} \mid \mathsf{true}\}\to\{x{:}\mathsf{Int} \mid x > 0\}$. For brevity, we refer to the domains as $T_{i\,1}$ and the codomains as $T_{i\,2}$. We find that $(\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ v_1)\ v_2$ steps in classic λ_{H} to:

$$\begin{array}{l}\langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l \\ \quad (v_1\ (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\ v_2))\end{array}$$

$$
\begin{array}{l}
e = \langle \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \rangle^{l_2} \\
\quad\; (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1})) \\
\hfill (\text{E\_COERCE}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\{x:\mathsf{Int} \mid x \neq 0\}^{l_3}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \rangle^{l_2} \\
\quad\; (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1})) \\
\hfill (\text{E\_CASTINNER/E\_COERCE}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\{x:\mathsf{Int} \mid x \neq 0\}^{l_3}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\{x:\mathsf{Int} \mid x \bmod 2 = 0\}^{l_2}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \rangle^{l_2} \\
\quad\; (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1})) \\
\hfill (\text{E\_CASTMERGEE}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{r'}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1}) \\
\text{where } r' = \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}^{l_2}, \{x{:}\mathsf{Int} \mid x \neq 0\}^{l_3} \\
\hfill (\text{E\_CASTINNER/E\_COERCE}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{r'}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\{x:\mathsf{Int} \mid x \geq 0\}^{l_1}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1}) \\
\hfill (\text{E\_CASTMERGEE}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{r}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3}\ {-1} \\
\text{where } r = \{x{:}\mathsf{Int} \mid x \geq 0\}^{l_1}, r' \\
\hfill (\text{E\_COERCESTACK}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid x \neq 0\}, ?, r, -1, -1 \rangle^{\bullet} \\
\hfill (\text{E\_STACKPOP}) \\
\longrightarrow_{\mathsf{E}} \langle \{x{:}\mathsf{Int} \mid x \neq 0\}, ?, r', -1, \\
\quad \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, -1 \geq 0, -1 \rangle^{l_1} \rangle^{\bullet} \\
\longrightarrow_{\mathsf{E}}^{*} \Uparrow l_1
\end{array}
$$

Figure 5. Example of eidetic λ_{H}

Note that T_1's domain is checked but its codomain isn't; the reverse is true for T_2. When looking at a cast, we can read off which refinements are checked by looking at the positive parts of the target type and the negative parts of the source type. The relationship between casts and polarity is not a new one [4, 9, 13, 15, 31]. Unlike casts, coercions directly express the sequence of checks to be performed. Consider the coercion generated from the cast above, recalling that $T_{i\,1}$ and $T_{i\,2}$ are the domains and codomains of T_1 and T_2:

$$
\begin{array}{ll}
 & (\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^{l}\ v_1)\ v_2 \\
\longrightarrow_{\mathsf{E}} & (\langle T_1 \overset{c}{\Rightarrow} T_2 \rangle^{\bullet}\ v_1)\ v_2 \\
 & \quad \text{where } c = \{x{:}\mathsf{Int} \mid x \geq 0\}^{l} \mapsto \{x{:}\mathsf{Int} \mid x > 0\}^{l} \\
\longrightarrow_{\mathsf{E}} & (\langle T_{11} \to T_{12} \overset{c}{\Rightarrow} T_{21} \to T_{22} \rangle^{\bullet}\ v_1)\ v_2 \\
\longrightarrow_{\mathsf{E}} & \langle T_{12} \overset{\{x:\mathsf{Int} \mid x > 0\}^{l}}{\Rightarrow} T_{22} \rangle^{\bullet}\ (v_1\ (\langle T_{21} \overset{\{x:\mathsf{Int} \mid x \geq 0\}^{l}}{\Rightarrow} T_{11} \rangle^{\bullet}\ v_2))
\end{array}
$$

In this example, there is only a single blame label, l. Tracking blame labels is critical for exactly matching classic λ_{H}'s behavior. The examples rely on $\supset$ being reflexive. First, we return to our example from before in Figure 5. Throughout the merging, each refinement type retains its own original blame label, allowing eidetic λ_{H} to behave just like classic λ_{H}.

We offer a final pair of examples, showing how coercions with redundant types are merged. The intuition here is that positive positions are checked covariantly—oldest (innermost) cast first—while negative positions are checked contravariantly—newest (outermost) cast first. Consider the classic λ_{H} term:

$$
\begin{array}{rl}
T_1 &= \{x{:}\mathsf{Int} \mid e_{11}\} \to \{x{:}\mathsf{Int} \mid e_{21}\} \\
T_2 &= \{x{:}\mathsf{Int} \mid e_{12}\} \to \{x{:}\mathsf{Int} \mid e_{22}\} \\
T_3 &= \{x{:}\mathsf{Int} \mid e_{13}\} \to \{x{:}\mathsf{Int} \mid e_{22}\} \\
e &= \langle T_2 \overset{\bullet}{\Rightarrow} T_3 \rangle^{l_2}\ (\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^{l_1}\ v)
\end{array}
$$

Note that the casts run inside-out, from old to new in the positive position, but they run from the outside-in, new to old, in the negative position.

$$
\begin{array}{l}
e\ v' \longrightarrow_{\mathsf{C}} \langle \{x{:}\mathsf{Int} \mid e_{22}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{22}\} \rangle^{l_2} \\
\qquad (\langle \{x{:}\mathsf{Int} \mid e_{21}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{22}\} \rangle^{l_1} \\
\qquad\quad (v\ (\langle \{x{:}\mathsf{Int} \mid e_{12}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{12}\} \rangle^{l_1} \\
\qquad\qquad (\langle \{x{:}\mathsf{Int} \mid e_{13}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{12}\} \rangle^{l_2}\ v'))))
\end{array}
$$

The key observation for eliminating redundant checks is that only the check run first can fail—there's no point in checking a predicate contract twice on the same value. So eidetic λ_{H} merges like so:

$$
\begin{array}{l}
e \longrightarrow_{\mathsf{E}}^{*} \langle T_2 \overset{\{x:\mathsf{Int} \mid e_{12}\}^{l_2} \mapsto \{x:\mathsf{Int} \mid e_{22}\}^{l_2}}{\Rightarrow} T_3 \rangle^{\bullet} \\
\qquad (\langle T_1 \overset{\{x:\mathsf{Int} \mid e_{11}\}^{l_1} \mapsto \{x:\mathsf{Int} \mid e_{22}\}^{l_1}}{\Rightarrow} T_2 \rangle^{\bullet}\ v) \\
\quad \longrightarrow_{\mathsf{E}} \langle T_1 \overset{c}{\Rightarrow} T_3 \rangle^{\bullet}\ v
\end{array}
$$

where

$$
\begin{array}{rl}
c = & \mathsf{join}(\{x{:}\mathsf{Int} \mid e_{12}\}^{l_2}, \{x{:}\mathsf{Int} \mid e_{11}\}^{l_1}) \mapsto \\
 & \mathsf{join}(\{x{:}\mathsf{Int} \mid e_{22}\}^{l_1}, \{x{:}\mathsf{Int} \mid e_{22}\}^{l_2}) \\
= & \{x{:}\mathsf{Int} \mid e_{12}\}^{l_2}, \{x{:}\mathsf{Int} \mid e_{11}\}^{l_1} \mapsto \{x{:}\mathsf{Int} \mid e_{22}\}^{l_1}
\end{array}
$$

The coercion merge operator eliminates the redundant codomain check, choosing to keep the one with blame label l_1. Choosing l_1 makes sense here because the codomain is a positive position and l_1 is the older, innermost cast. We construct a similar example for merges in negative positions.

$$
\begin{array}{rl}
T_1 &= \{x{:}\mathsf{Int} \mid e_{11}\} \to \{x{:}\mathsf{Int} \mid e_{21}\} \\
T_2' &= \{x{:}\mathsf{Int} \mid e_{11}\} \to \{x{:}\mathsf{Int} \mid e_{22}\} \\
T_3' &= \{x{:}\mathsf{Int} \mid e_{13}\} \to \{x{:}\mathsf{Int} \mid e_{23}\} \\
e' &= \langle T_2' \overset{\bullet}{\Rightarrow} T_3' \rangle^{l_2}\ (\langle T_1 \overset{\bullet}{\Rightarrow} T_2' \rangle^{l_1}\ v)
\end{array}
$$

Again, the unfolding runs the positive parts inside-out and the negative parts outside-in when applied to a value v':

$$
\begin{array}{l}
\langle \{x{:}\mathsf{Int} \mid e_{22}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{23}\} \rangle^{l_2} \\
\quad (\langle \{x{:}\mathsf{Int} \mid e_{21}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{22}\} \rangle^{l_1} \\
\quad\quad (v\ (\langle \{x{:}\mathsf{Int} \mid e_{11}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{11}\} \rangle^{l_1} \\
\quad\quad\quad (\langle \{x{:}\mathsf{Int} \mid e_{13}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid e_{11}\} \rangle^{l_2}\ v'))))
\end{array}
$$

Running the example in eidetic λ_{H}, we reduce the redundant checks in the domain:

$$
\begin{array}{l}
e' \longrightarrow_{\mathsf{E}}^{*} \langle T_2' \overset{\{x:\mathsf{Int} \mid e_{11}\}^{l_2} \mapsto \{x:\mathsf{Int} \mid e_{23}\}^{l_2}}{\Rightarrow} T_3' \rangle^{\bullet} \\
\qquad (\langle T_1 \overset{\{x:\mathsf{Int} \mid e_{11}\}^{l_1} \mapsto \{x:\mathsf{Int} \mid e_{22}\}^{l_1}}{\Rightarrow} T_2' \rangle^{\bullet}\ v) \\
\quad \longrightarrow_{\mathsf{E}} \langle T_1 \overset{c}{\Rightarrow} T_3' \rangle^{\bullet}\ v
\end{array}
$$

where

$$
\begin{array}{rl}
c = & \mathsf{join}(\{x{:}\mathsf{Int} \mid e_{11}\}^{l_2}, \{x{:}\mathsf{Int} \mid e_{11}\}^{l_1}) \mapsto \\
 & \mathsf{join}(\{x{:}\mathsf{Int} \mid e_{22}\}^{l_1}, \{x{:}\mathsf{Int} \mid e_{23}\}^{l_2}) \\
= & \{x{:}\mathsf{Int} \mid e_{12}\}^{l_2} \mapsto \{x{:}\mathsf{Int} \mid e_{22}\}^{l_1}, \{x{:}\mathsf{Int} \mid e_{23}\}^{l_2}
\end{array}
$$

Following the outside-in rule for negative positions, we keep the blame label l_2 from the newer, outermost cast.

4.1 Metatheory

The proof of type soundness is a standard syntactic proof, relying on a few small lemmas concerning refinement list well formedness and the generic metatheory described in Section 3.3. The full proofs are in Appendix A.3.

Lemma [Eidetic canonical forms (A.15)]: If $\emptyset \vdash_{\mathsf{E}} e : T$ and $\mathsf{val}_{\mathsf{E}}\ e$ then:

– If $T = \{x{:}B \mid e'\}$, then $e = k$ and $\mathsf{ty}(k) = B$ and $e'[e/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true}$.
– If $T = T_{21}\to T_{22}$, then either $e = \lambda x{:}T.\ e'$ or $e = \langle T_{11}\to T_{12} \stackrel{c_1\mapsto c_2}{\Rightarrow} T_{21}\to T_{22}\rangle^{\bullet}\ \lambda x{:}T_{11}.\ e'$.

Lemma [Eidetic progress (A.16)]: If $\emptyset \vdash_{\mathsf{E}} e : T$, then either:

1. $\mathsf{result}_{\mathsf{E}}\ e$, i.e., $e = \Uparrow l$ or $\mathsf{val}_{\mathsf{E}}\ e$; or
2. there exists an e' such that $e \longrightarrow_{\mathsf{E}} e'$.

Lemma [Eidetic preservation (A.20)]: If $\emptyset \vdash_{\mathsf{E}} e : T$ and $e \longrightarrow_{\mathsf{E}} e'$ then $\emptyset \vdash_{\mathsf{E}} e' : T$.

Eidetic λ_{H} shares source programs (Definition 3.1) with classic λ_{H}. We can therefore say that classic and eidetic λ_{H} are really just *modes* of a single language.

Lemma [Source program typing for eidetic λ_{H} (A.21)]: Source programs are well typed in C iff they are well typed in E, i.e.:

– $\Gamma \vdash_{\mathsf{C}} e : T$ as a source program iff $\Gamma \vdash_{\mathsf{E}} e : T$ as a source program.
– $\vdash_{\mathsf{C}} T$ as a source program iff $\vdash_{\mathsf{E}} T$ as a source program.
– $\vdash_{\mathsf{C}} \Gamma$ as a source program iff $\vdash_{\mathsf{E}} \Gamma$ as a source program.

5. Soundness for space efficiency

We want space efficiency to be *sound*: it would be space efficient to never check anything. Classic λ_{H} is normative: the more a mode behaves like classic λ_{H}, the "sounder" it is.

A single property summarizes how a space-efficient calculus behaves with respect to classic λ_{H}: cast congruence. In classic λ_{H}, if $e_1 \longrightarrow_{\mathsf{C}} e_2$ then $\langle T_1 \stackrel{\bullet}{\Rightarrow} T_2\rangle^l\ e_1$ and $\langle T_1 \stackrel{\bullet}{\Rightarrow} T_2\rangle^l\ e_2$ behave identically. This cast congruence principle is easy to see, because E_CastInnerC applies freely. In eidetic λ_{H}, however, E_CastInner can only apply when E_CastMergeE doesn't. Merged casts may not behave the same as running the two casts separately. Eidetic λ_{H} recovers a complete cast congruence, just like classic λ_{H} has. Diagrammatically:

$$\begin{tikzcd}
 & e_1 \arrow[r, "\mathsf{E}"] & e_2 & \\
 & & \Downarrow & \\
\langle T_1 \stackrel{\bullet}{\Rightarrow} T_2\rangle^l\ e_1 \arrow[dr, "*"'] & & & \langle T_1 \stackrel{\bullet}{\Rightarrow} T_2\rangle^l\ e_2 \arrow[dl, "*"] \\
 & \mathsf{result}_{\mathsf{E}}\ e & &
\end{tikzcd}$$

The proof is in Appendix B, but it is worth observing here that eidetic λ_{H} needs a proof of idempotency to justify the way it uses reflexivity to eliminate redundant coercions: checking a property once is as good as checking it twice. Naturally, this property only holds without state.

Our proofs relating classic λ_{H} and eidetic λ_{H} are by logical relations, found in Figure 6. In the extended version, the soundness proofs for all three different space-efficient modes use a single mode-indexed logical relation. Here we give its restriction to eidetic λ_{H}. As far as alternative techniques go, an induction over evaluation derivations wouldn't give us enough information about evaluations that return lambda abstractions. Other contextual equivalence techniques (e.g., bisimulation) would probably work, too.

Value rules $\boxed{e_1 \sim_{\mathsf{E}} e_2 : T}$

$$\begin{array}{rcl}
k \sim_{\mathsf{E}} k : \{x{:}B \mid e\} & \iff & \mathsf{ty}(k) = B \wedge e[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true} \\
e_{11} \sim_{\mathsf{E}} e_{21} : T_1 \to T_2 & \iff & \mathsf{val}_{\mathsf{C}}\ e_1 \wedge \mathsf{val}_{\mathsf{E}}\ e_2 \wedge \\
 & & \forall e_{12} \sim_{\mathsf{E}} e_{22} : T_1.\ e_{11}\ e_{12} \simeq_{\mathsf{E}} e_{21}\ e_{22} : T_2
\end{array}$$

Term rules $\boxed{e_1 \simeq_{\mathsf{E}} e_2 : T}$

$$\begin{array}{c}
e_1 \simeq_{\mathsf{E}} e_2 : T \\
\iff \\
\begin{pmatrix} e_1 \longrightarrow^*_{\mathsf{C}} \Uparrow l \wedge \\ e_2 \longrightarrow^*_{\mathsf{E}} \Uparrow l \end{pmatrix} \vee \begin{pmatrix} e_1 \longrightarrow^*_{\mathsf{C}} e_1' \wedge \mathsf{val}_{\mathsf{C}}\ e_1' \wedge \\ e_2 \longrightarrow^*_{\mathsf{E}} e_2' \wedge \mathsf{val}_{\mathsf{E}}\ e_2' \wedge \\ e_1' \sim_{\mathsf{E}} e_2' : T \end{pmatrix}
\end{array}$$

Type rules $\boxed{T_1 \sim_{\mathsf{E}} T_2}$

$$\begin{array}{rcl}
\{x{:}B \mid e_1\} \sim_{\mathsf{E}} \{x{:}B \mid e_2\} & \iff & \\
\multicolumn{3}{l}{\forall e_1' \sim_{\mathsf{E}} e_2' : \{x{:}B \mid \mathsf{true}\}.\ e_1[e_1'/x] \simeq_{\mathsf{E}} e_2[e_2'/x] : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}} \\
T_{11}\to T_{12} \sim_{\mathsf{E}} T_{21}\to T_{22} & \iff & T_{11} \sim_{\mathsf{E}} T_{21} \wedge T_{12} \sim_{\mathsf{E}} T_{22}
\end{array}$$

Closing substitutions and open terms $\boxed{\Gamma \models_{\mathsf{E}} \delta}$

$\boxed{\Gamma \vdash e_1 \simeq_{\mathsf{E}} e_2 : T}$

$$\begin{array}{rcl}
\Gamma \models_{\mathsf{E}} \delta & \iff & \forall x \in \mathsf{dom}(\Gamma).\ \delta_1(x) \sim_{\mathsf{E}} \delta_2(x) : \Gamma(x) \\
\Gamma \vdash e_1 \simeq_{\mathsf{E}} e_2 : T & \iff & \forall \Gamma \models_{\mathsf{E}} \delta.\ \delta_1(e_1) \simeq_{\mathsf{E}} \delta_2(e_2) : T
\end{array}$$

Figure 6. Blame-exact, symmetric logical relation between classic λ_{H} and eidetic λ_{H}

Mode	Cast size	Pending casts
Classic ($m = \mathsf{C}$)	$2W_h + L$	∞
Eidetic ($m = \mathsf{E}$)	$s2^{L+W_B}$	$\lvert e\rvert$

Table 1. Space efficiency of λ_{H}

Lemma [Similar casts are logically related (B.3)]: If $T_1 \sim_{\mathsf{E}} T_1'$ and $T_2 \sim_{\mathsf{E}} T_2'$ and $e_1 \sim_{\mathsf{E}} e_2 : T_1$, then $\langle T_1 \stackrel{\bullet}{\Rightarrow} T_2\rangle^l\ e_1 \simeq_{\mathsf{E}} \langle T_1' \stackrel{\bullet}{\Rightarrow} T_2'\rangle^l\ e_2 : T_2$.

Lemma [Relating classic and eidetic source programs (B.4)]:

1. If $\Gamma \vdash_{\mathsf{C}} e : T$ as a source program then $\Gamma \vdash e \simeq_{\mathsf{E}} e : T$.
2. If $\vdash_{\mathsf{C}} T$ as a source program then $T \sim_{\mathsf{E}} T$.

6. Bounds for space efficiency

We have claimed that eidetic λ_{H} is space efficient: what do we mean? What sort of space efficiency have we achieved? We summarize the results in Table 1; proofs are in Appendix C. From a high level, there are only a finite number of types that appear in our programs, and this set of types can only reduce as the program runs. We can effectively code each type in the program as an integer, allowing us to efficiently run the $\supset$ predicate.

Suppose that a type of height h can be represented in W_h bits and a label in L bits. (Type heights are defined in Figure 7 in Appendix C.) Casts in classic λ_{H} each take up $2W_h + L$ bits: two types and a blame label. Coercions in eidetic λ_{H} have a different form: the only types recorded are those of height 1, i.e., refinements of base types. Pessimistically, each of these may appear at every position in a function coercion $c_1 \mapsto c_2$. We use s to indicate the "size" of a function type, i.e., the number of positions it has. As a first pass, a set of refinements and blame labels take up 2^{L+W_1} space. But in fact these coercions must all be between refinements *of the same base type*, leading to 2^{L+W_B} space per coercion, where W_B is the highest number of refinements of any single base type. We now have our worst-case space complexity: $s2^{L+W_B}$. A more precise bound might track which refinements appear in which parts of a function type, but in the worst case—each refinement appears in every position—it degenerates to the bound we give here. Classic λ_{H} can have an infinite number of "pending casts"—casts and function proxies—in a program. Eidetic λ_{H} can have no more than one pending cast per term node. Abstractions are limited to a single function proxy, and E_CASTMERGEE merges adjacent pending casts.

The text of a program e is finite, so the set of types appearing in the program, $\mathsf{types}(e)$, is also finite. Since reduction doesn't introduce types, we can bound the number of types in a program (and therefore the size of casts). We can therefore fix a numerical coding for types at runtime, where we can encode a type in $W = \log_2(|\mathsf{types}(e)|)$ bits. In a given cast, W over-approximates how many types can appear: the source, target, and annotation must all be compatible, which means they must also be of the same height. We can therefore represent the types in casts with fewer bits: $W_h = \log_2(|\{T \mid T \in \mathsf{types}(e) \wedge \mathsf{height}(T) = h\}|)$. In the worst case, we revert to the original bound: all types in the program are of height 1. Even so, there are never casts between different base types B and B', so $W_B = \max_B \log_2(|\{\{x{:}B \mid e\} \in \mathsf{types}(e)\}|)$. Eidetic λ_{H}'s coercions never hold types greater than height 1. The types on its casts are erasable once the coercions are generated, because coercions drive the checking.

6.1 Representation choices

The bounds we find here are *galactic*. Having established that contracts are theoretically space efficient, making an implementation practically space efficient is a different endeavor, involving careful choices of representations and calling conventions.

Eidetic λ_{H}'s space bounds rely only on the reflexivity of the $\supset$ predicate, since we leave it abstract. We have identified one situation where the relation allows us to find better space bounds: mutual implication.

If $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}$ and $\{x{:}B \mid e_2\} \supset \{x{:}B \mid e_1\}$, then these two types are equivalent, and only one ever need be checked. Which to check could be determined by a compiler with a suitably clever cost model. Note that our proofs don't entirely justify this optimization. By default, our join operator will take whichever of $\{x{:}B \mid e_1\}$ and $\{x{:}B \mid e_2\}$ was meant to be checked first. Adapting join to always choose one based on some preference relation would not be particularly hard, and we believe the proofs adapt easily.

Other analyses of the relation seem promising at first, but in fact do not allow more compact representations. Suppose we have a program where $\{x{:}B \mid e_1\} \supset \{x{:}B \mid e_2\}$ but not vice versa, and that B is our worst case type. That is, $W_B = 2$, because there are 2 different refinements of B and fewer refinements of other base types. The worst-case representation for a refinement list is 2 bits, with bit b_i indicating whether e_i is present in the list. Can we do any better than 2 bits, since e_1 can stand in for e_2? Could we represent the two types as just 1 bit? We cannot when (a) there are constants that pass one type but not the other and (b) when refinement lists are in the reverse order of implication. Suppose there is some k such that $e_2[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true}$ and $e_1[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{false}$. Now we consider a concatenation of refinement lists in the reverse ordering: $\mathsf{join}(\{x{:}B \mid e_2\}^l, \{x{:}B \mid e_1\}^{l'})$. We must retain both checks, since different failures lead to different blame. The k that passes e_2 but not e_1 should raise $\Uparrow l'$, but other k' that fail for both types should raise $\Uparrow l$. One bit isn't enough to capture the situation of having the coercion $\{x{:}B \mid e_2\}^l, \{x{:}B \mid e_1\}^{l'}$.

Finally, what is the right representation for a function? When calling a function, do we need to run coercions or not? Jeremy Siek suggested a "smart closure" which holds the logic for branching inside its own code; this may support better branch prediction than an indirect jump or branching at call sites.

7. Related work

Some earlier work uses first-class casts, whereas our casts are always applied to a term [1, 17]. It is of course possible to η-expand a cast with an abstraction, so no expressiveness is lost. Leaving casts fully applied saves us from the puzzling rules managing how casts work on other casts in space-efficient semantics, like: $\langle T_{11} \to T_{12} \overset{\bullet}{\Rightarrow} T_{21} \to T_{22} \rangle^l \, \langle T_{11} \overset{\bullet}{\Rightarrow} T_{12} \rangle^{l'} \longrightarrow_{\mathsf{F}} \langle T_{21} \overset{\bullet}{\Rightarrow} T_{22} \rangle^l$.

Previous approaches to space-efficiency have focused on gradual typing [27]. This work uses coercions [14], casts, casts annotated with intermediate types a/k/a *threesomes*, or some combination of all three [8, 16, 24, 26, 28]. Recent work relates all three frameworks, making particular use of coercions [25]. Our type structure differs from that of gradual types, so our space bounds come in a somewhat novel form. Gradual types, without the more complicated checking that comes with predicate contracts, allow for simpler designs. Siek and Wadler [28] can define a simple recursive operator on labeled types with a strong relationship to subtyping, the fundamental property of casts. We haven't been able to discover a connection in our setting. Instead, we ignore the type structure of functions and focus our attention on managing labels in lists of first-order predicate contracts. In the gradual world, only Rastogi et al. [23] take a similar approach, "recursively deconstruct[ing] higher-order types down to their first-order parts" when they compute the closure of flows into and out of type variables. Gradual types occasionally have simpler proofs, too, e.g., by induction on evaluation [24]; even when strong reasoning principles are needed, the presence of dynamic types leads them to use bisimulation [8, 25, 28]. We use logical relations because λ_{H}'s type structure is readily available, and because they allow us to easily reason about how checks evaluate.

Our coercions are inspired by Henglein's coercions for modeling injection to and projection from the dynamic type [14]. Henglein's primitive coercions tag and untag values, while ours represent checks to be performed on base types; both our formulation and Henglein's use structural function coercions.

Greenberg [9], the most closely related work, offers a coercion language combining the dynamic types of Henglein's original work with predicate contracts; his EFFICIENT language does not quite achieve "sound" space efficeincy. Rather, it is forgetful, occasionally dropping casts. He omits blame, though he conjectures that blame for coercions reads left to right (as it does in Siek and Garcia [26]); our eidetic λ_{H} verifies this conjecture. While Greenberg's languages offer dynamic, simple, and refined types, our types here are entirely refined. His coercions use Henglein's ! and ? syntax for injection and projection, while our coercions lack such a distinction. In our refinement lists, each coercion simultaneously projects from one refinement type and injects into another (possibly producing blame). We reduce notation by omitting the interrobang '‽'.

Dimoulas et al. [3] introduce *option contracts*, which offer a programmatic way of turning off contract checking, as well as a controlled way to "pass the buck", handing off contracts from component to component. Option contracts address time efficiency, not space efficiency. Findler et al. [6] studied space and time efficiency for datatype contracts, as did Koukoutos and Kuncak [19].

Racket contracts have a mild form of space efficiency: the `tail-marks-match?` predicate[1] checks for exact duplicate contracts and blame at tail positions. The redundancy it detects seems

[1] From `racket/collects/racket/contract/private/arrow.rkt`.

to rely on pointer equality. Since Racket contracts are (a) module-oriented "macro" contracts and (b) first class, this optimization is somewhat unpredictable—and limited compared with our eidetic calculus, which can handle differing contracts and blame labels.

8. Conclusion and future work

Semantics-preserving space efficiency for manifest contracts is possible—leaving the admissibility of state as the final barrier to practical utility. We established that eidetic λ_{H} behaves exactly like its classic counterpart without compromising space usage.

We believe it would be easy to design a latent version of eidetic λ_{H}, following the translations in Greenberg et al. [11].

In our simple (i.e., not dependent) case, our refinement types close over a single variable of base type. Space efficiency for a dependent calculus remains open. The first step towards dependent types would be extending $\supset$ with a context (and a source of closing substitutions, a serious issue [1]). In a dependent setting the definition of what it means to compare closures isn't at all clear. Closures' environments may contain functions, and closures over extensionally equivalent functions may not be intensionally equal. A more nominal approach to contract comparison may resolve some of the issues here. Comparisons might be more straightforward when contracts are explicitly declared and referenced by name. Similarly, a dependent $\supset$ predicate might be more easily defined over some explicit structured family of types, like a lattice. Findler et al. [6] has made some progress in this direction.

Finally, a host of practical issues remain. Beyond representation choices, having expensive checks makes it important to predict *when* checks happen. The $\supset$ predicate compares closures and will surely have delicate interactions with optimizations.

Acknowledgments

Comments from Rajeev Alur, Ron Garcia, Fritz Henglein, Greg Morrisett, Stephanie Weirich, Phil Wadler, and Steve Zdancewic improved a previous version of this work done at the University of Pennsylvania. Some comments from Benjamin Pierce led me to realize that a cast formulation was straightforward. Discussions with Atsushi Igarashi, Robby Findler, and Sam Tobin-Hochstadt greatly improved the quality of the exposition. Phil Wadler encouraged me to return to coercions to understand the eidetic formulation. The POPL reviewers had many excellent suggestions, and Robby Findler helped once more with angelic guidance. Hannah de Keijzer proofread the paper.

This work was supported in part by the NSF under grants TC 0915671 and SHF 1016937 and 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 the author's and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

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

[2] 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.

[3] C. Dimoulas, R. Findler, and M. Felleisen. Option contracts. In *OOPSLA*, pages 475 – 494, 2013.

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

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

[6] R. B. Findler, S.-Y. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Implementation and Application of Functional Languages*, pages 111–128. 2008. .

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

[8] R. Garcia. Calculating threesomes, with blame. In *International Conference on Functional Programming (ICFP)*, 2013.

[9] M. Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, November 2013.

[10] M. Greenberg. Space-efficient manifest contracts, 2014. URL `http://arxiv.org/abs/1410.2813`. Technical report.

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

[12] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. *Journal of Functional Programming (JFP)*, 22(3):225–274, May 2012.

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

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

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

[16] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 23(2):167–189, June 2010. .

[17] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Prog. Lang. Syst.*, 32:6:1–6:34, 2010.

[18] 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.

[19] E. Koukoutos and V. Kuncak. Checking data structure properties orders of magnitude faster. In *Runtime Verification*, pages 263–268. 2014. .

[20] R. Lipton, October 2010. URL `http://goo.gl/6GrgtO`.

[21] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.

[22] PLT. Racket contract system, 2013. URL `http://pre.plt-scheme.org/docs/html/guide/contracts.html`.

[23] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Principles of Programming Languages (POPL)*, 2012. .

[24] J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *Programming Languages and Systems*, volume 5502 of *LNCS*, pages 17–31. 2009. .

[25] J. Siek, P. Thiemann, and P. Wadler. Blame, coercion, and threesomes: Together again for the first time. 2015. URL `http://homepages.inf.ed.ac.uk/wadler/topics/blame.html#coercions`.

[26] J. G. Siek and R. Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming (SFP)*, 2012.

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

[28] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Principles of Programming Languages (POPL)*, pages 365–376, 2010. .

[29] 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, 2009. ISBN 978-1-60558-332-7. .

[30] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA*, 2006. .

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

[32] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

A. Proofs of type soundness

This appendix includes the proofs of type soundness for all four modes of λ_{H}; we first prove some universally applicable metatheoretical properties.

A.1 Generic metatheory

A.1 Lemma [Weakening]: If $\Gamma_1, \Gamma_2 \vdash_m e : T$ and $\vdash_m T'$ and x is fresh, then $\vdash_m \Gamma_1, x{:}T', \Gamma_2$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash_m e : T$.

A.2 Lemma [Substitution]: If $\Gamma_1, x{:}T', \Gamma_2 \vdash_m e : T$ and $\emptyset \vdash_m e' : T'$, then $\Gamma_1, \Gamma_2 \vdash_m e[e'/x] : T$ and $\vdash_m \Gamma_1, \Gamma_2$.

A.3 Lemma [Regularity]: If $\Gamma \vdash_m e : T$, then $\vdash_m \Gamma$ and $\vdash_m T$.

A.4 Lemma [Similarity is reflexive]: If $\vdash T \parallel T$.

Proof: By induction on T.

($T = \{x{:}B \mid e\}$) By S_REFINE.
($T = T_1 \rightarrow T_2$) By S_FUN and the IHs.

$\square$

A.5 Lemma [Similarity is symmetric]: If $\vdash T_1 \parallel T_2$, then $\vdash T_2 \parallel T_1$.

Proof: By induction on the similarity derivation.

(S_REFINE) By S_REFINE.
(S_FUN) By S_FUN and the IHs.

$\square$

A.6 Lemma [Similarity is transitive]: If $\vdash T_1 \parallel T_2$ and $\vdash T_2 \parallel T_3$, then $\vdash T_1 \parallel T_3$.

Proof: By induction on the derivation of $\vdash T_1 \parallel T_2$.

(S_REFINE) The other derivation must also be by S_REFINE; by S_REFINE.
(S_FUN) The other derivation must also be by S_FUN; by S_FUN and the IHs.

$\square$

A.7 Lemma [Well formed type sets have similar indices]: If $\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2$ then $\vdash T_1 \parallel T_2$.

Proof: Immediate, by inversion. $\square$

A.8 Lemma [Type set well formedness is symmetric]: $\vdash_m a \parallel T_1 \Rightarrow T_2$ iff $\vdash_m a \parallel T_2 \Rightarrow T_1$ for all $m \neq \mathsf{E}$.

Proof: We immediately have $\vdash_m T_1$ and $\vdash_m T_2$, and $\vdash T_1 \parallel T_2$ iff $\vdash T_2 \parallel T_1$ by Lemma A.5.

If $m = \mathsf{C}$ or $m = \mathsf{F}$, then by A_NONE and symmetry of similarity (Lemma A.5.

If $m = \mathsf{H}$, then let $T \in \mathcal{S}$ be given. The $\vdash_{\mathsf{H}} T$ premises hold immediately; we are then done by transitivity (Lemma A.6) and symmetry (Lemma A.5) of similarity ($\vdash T \parallel T_1$ iff $\vdash T \parallel T_2$ when $\vdash T_1 \parallel T_2$). $\square$

A.9 Lemma [Type set well formedness is transitive]: If $\vdash T_1 \parallel T_2$ and $\vdash_m a \parallel T_2 \Rightarrow T_3$ and $\vdash_m T_1$ and $m \neq \mathsf{E}$ then $\vdash_m a \parallel T_1 \Rightarrow T_3$.

Proof: We immediately have $\vdash_m T_1$ and $\vdash_m T_3$; we have $\vdash T_1 \parallel T_3$ by transitivity of similarity (Lemma A.6).

If $m = \mathsf{C}$ or $m = \mathsf{F}$, we are done immediately by A_NONE.

If, on the other hand, $m = \mathsf{H}$, let $T \in \mathcal{S}$ be given. We know that $\vdash_{\mathsf{H}} T$ and $\vdash T \parallel T_2$; by symmetry (Lemma A.5) and transitivity (Lemma A.6) of similarity, we are done by A_TYPESET. $\square$

A.2 Classic type soundness

A.10 Lemma [Classic determinism]: If $e \longrightarrow_{\mathsf{C}} e_1$ and $e \longrightarrow_{\mathsf{C}} e_2$ then $e_1 = e_2$.

Proof: By induction on the first evaluation derivation. $\square$

A.11 Lemma [Classic canonical forms]: If $\emptyset \vdash_{\mathsf{C}} e : T$ and $\mathsf{val}_{\mathsf{C}}\ e$ then:

– If $T = \{x{:}B \mid e'\}$, then $e = k$ and $\mathsf{ty}(k) = B$ and $e'[e/x] \longrightarrow^*_{\mathsf{C}} \mathsf{true}$.
– If $T = T_1 \rightarrow T_2$, then either $e = \lambda x{:}T.\ e'$ or $e = \langle T_{11} \rightarrow T_{12} \overset{\bullet}{\Rightarrow} T_{21} \rightarrow T_{22} \rangle^l\ e'$.

A.12 Lemma [Classic progress]: If $\emptyset \vdash_{\mathsf{C}} e : T$, then either:

1. $\mathsf{result}_{\mathsf{C}}\ e$, i.e., $e = \Uparrow l$ or $\mathsf{val}_{\mathsf{C}}\ e$; or
2. there exists an e' such that $e \longrightarrow_{\mathsf{C}} e'$.

Proof: By induction on the typing derivation. $\square$

A.13 Lemma [Classic preservation]: If $\emptyset \vdash_{\mathsf{C}} e : T$ and $e \longrightarrow_{\mathsf{C}} e'$, then $\emptyset \vdash_{\mathsf{C}} e' : T$.

Proof: By induction on the typing derivation. $\square$

A.3 Eidetic type soundness

A.14 Lemma [Determinism of eidetic λ_{H}]: If $e \longrightarrow_{\mathsf{E}} e_1$ and $e \longrightarrow_{\mathsf{E}} e_2$ then $e_1 = e_2$.

Proof: By induction on the first evaluation derivation. In every case, only a single step can be taken. $\square$

A.15 Lemma [Eidetic canonical forms]: If $\emptyset \vdash_{\mathsf{E}} e : T$ and $\mathsf{val}_{\mathsf{E}}\ e$ then:

– If $T = \{x{:}B \mid e'\}$, then $e = k$ and $\mathsf{ty}(k) = B$ and $e'[e/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true}$.
– If $T = T_{21} \rightarrow T_{22}$, then either $e = \lambda x{:}T.\ e'$ or $e = \langle T_{11} \rightarrow T_{12} \overset{c_1 \mapsto c_2}{\Rightarrow} T_{21} \rightarrow T_{22} \rangle^{\bullet}\ \lambda x{:}T_{11}.\ e'$.

A.16 Lemma [Eidetic progress]: If $\emptyset \vdash_{\mathsf{E}} e : T$, then either:

1. $\mathsf{result}_{\mathsf{E}}\ e$, i.e., $e = \Uparrow l$ or $\mathsf{val}_{\mathsf{E}}\ e$; or
2. there exists an e' such that $e \longrightarrow_{\mathsf{E}} e'$.

Proof: By induction on the typing derivation. $\square$

A.17 Lemma [Extended refinement lists are well formed]: If $\vdash_{\mathsf{E}} \{x{:}B \mid e\}$ and $\vdash_{\mathsf{E}} r \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}$ then $\vdash_{\mathsf{E}} \mathsf{join}(\{x{:}B \mid e\}^l, r) \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}$.

Proof: By cases on the rule used.

(A_REFINE) All of the premises are immediately restored except in one tricky case. When $\{x{:}B \mid e\} \supset \{x{:}B \mid e'\}$ where $\{x{:}B \mid e'\} \in r$ is the only type implying $\{x{:}B \mid e_2\}$. Then $\mathsf{drop}\,(r, \{x{:}B \mid e\})$ isn't well formed on its own, but adding $\{x{:}B \mid e\}^l$ makes it so by transitivity. If not, then we know that $\mathsf{drop}\,(r, \{x{:}B \mid e\})$ is well formed, and so is its extensions by assumption.

We know that there are no duplicates by reflexivity of $\supset$.
(A_FUN) Contradictory.

$\square$

A.18 Lemma [Merged coercions are well formed]: If $\vdash_{\mathsf{E}} c_1 \parallel T_1 \Rightarrow T_2$ and $\vdash_{\mathsf{E}} c_2 \parallel T_2 \Rightarrow T_3$ then $\vdash_{\mathsf{E}} \mathsf{join}(c_1, c_2) \parallel T_1 \Rightarrow T_3$.

Proof: By induction on c_1's typing derivation.

(A_REFINE) By the IH, Lemma A.17, and A_REFINE.
(A_FUN) By the IHs and A_FUN.

$\square$

A.19 Lemma [coerce generates well formed coercions]: If $\vdash T_1 \parallel T_2$ then $\vdash_{\mathsf{E}} \mathsf{coerce}(T_1, T_2, l) \parallel T_1 \Rightarrow T_2$.

Proof: By induction on the similarity derivation.

(S_REFINE) By A_REFINE, with $\mathsf{coerce}(\{x{:}B \mid e_1\}, \{x{:}B \mid e_2\}, l) = \{x{:}B \mid e_2\}^l$.
(S_FUN) By A_FUN and the IHs.

$\square$

A.20 Lemma [Eidetic preservation]: If $\emptyset \vdash_{\mathsf{E}} e : T$ and $e \longrightarrow_{\mathsf{E}} e'$ then $\emptyset \vdash_{\mathsf{E}} e' : T$.

Proof: By induction on the typing derivation. $\square$

A.21 Lemma [Source program typing for eidetic λ_{H}]: Source programs are well typed in C iff they are well typed in E, i.e.:

– $\Gamma \vdash_{\mathsf{C}} e : T$ as a source program iff $\Gamma \vdash_{\mathsf{E}} e : T$ as a source program.
– $\vdash_{\mathsf{C}} T$ as a source program iff $\vdash_{\mathsf{E}} T$ as a source program.
– $\vdash_{\mathsf{C}} \Gamma$ as a source program iff $\vdash_{\mathsf{E}} \Gamma$ as a source program.

Proof: By mutual induction on e, T, and Γ. $\square$

B. Proofs of space-efficiency soundness

B.1 Lemma [Idempotence of coercions]: If $\emptyset \vdash_{\mathsf{E}} k : \{x{:}B \mid e_1\}$ and $\vdash_{\mathsf{E}} \mathsf{join}(r_1, r_2) \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}$, then for all $\mathsf{result}_{\mathsf{E}}$ e, we have $\langle \{x{:}B \mid e_1\} \overset{\mathsf{join}(r_1, \mathsf{drop}(r_2, \{x:B \mid e_1\}))}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^\bullet\ k \longrightarrow^*_{\mathsf{E}} e$ iff $\langle \{x{:}B \mid e_1\} \overset{\mathsf{join}(r_1, r_2)}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^\bullet\ k \longrightarrow^*_{\mathsf{E}} e$.

Proof: By induction on their evaluation derivations: the only difference is that the latter derivation performs some extra checks that are implied by $e_1[k/x] \longrightarrow^*_{\mathsf{E}} \mathsf{true}$—which we already know to hold. $\square$

As before, cast congruence is the key lemma in our proof—in this case, the strongest property we have: reduction to identical results.

B.2 Lemma [Cast congruence (single step)]: If

– $\emptyset \vdash_{\mathsf{E}} e_1 : T_1$ and $\vdash_{\mathsf{E}} c \parallel T_1 \Rightarrow T_2$ (and so $\emptyset \vdash_{\mathsf{E}} \langle T_1 \overset{c}{\Rightarrow} T_2\rangle^\bullet\ e_1 : T_2$),
– $e_1 \longrightarrow_{\mathsf{E}} e_2$ (and so $\emptyset \vdash_{\mathsf{E}} e_2 : T_1$),

then for all $\mathsf{result}_{\mathsf{E}}$ e, we have $\langle T_1 \overset{c}{\Rightarrow} T_2\rangle^\bullet\ e_1 \longrightarrow^*_{\mathsf{E}} e$ iff $\langle T_1 \overset{c}{\Rightarrow} T_2\rangle^\bullet\ e_2 \longrightarrow^*_{\mathsf{E}} e$.

Proof: By cases on the step taken to find $e_1 \longrightarrow_{\mathsf{E}} e_2$. $\square$

Our proof strategy is as follows: we show that the casts between related types are applicative, and then we show that well typed source programs in classic λ_{H} are logically related to their translation. Our definitions are in Figure 6. Our logical relation is *blame-exact*. Like our proofs relating forgetful and heedful λ_{H} to classic λ_{H}, we use the space-efficient semantics in the refinement case and use space-efficient type indices.

B.3 Lemma [Similar casts are logically related]: If $T_1 \sim_{\mathsf{E}} T_1'$ and $T_2 \sim_{\mathsf{E}} T_2'$ and $e_1 \sim_{\mathsf{E}} e_2 : T_1$, then $\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e_1 \simeq_{\mathsf{E}} \langle T_1' \overset{\bullet}{\Rightarrow} T_2'\rangle^l\ e_2 : T_2$.

Proof: By induction on the invariant relation, using coercion congruence in the function case when e_2 is a function proxy. $\square$

B.4 Lemma [Relating classic and eidetic source programs]:

1. If $\Gamma \vdash_{\mathsf{C}} e : T$ as a source program then $\Gamma \vdash e \simeq_{\mathsf{E}} e : T$.
2. If $\vdash_{\mathsf{C}} T$ as a source program then $T \sim_{\mathsf{E}} T$.

Proof: By mutual induction on the typing derivations. $\square$

Term type extraction $\boxed{\mathsf{types}(e) : \mathcal{P}(T)}$

$$\begin{aligned}
\mathsf{types}(x) &= \emptyset \\
\mathsf{types}(k) &= \emptyset \\
\mathsf{types}(\lambda x{:}T.\ e) &= \mathsf{types}(T) \cup \mathsf{types}(e) \\
\mathsf{types}(\langle T_1 \overset{a}{\Rightarrow} T_2\rangle^l\ e) &= \mathsf{types}(T_1) \cup \mathsf{types}(T_2) \cup \\
&\quad\ \mathsf{types}(a) \cup \mathsf{types}(e) \\
\mathsf{types}(e_1\ e_2) &= \mathsf{types}(e_1) \cup \mathsf{types}(e_2) \\
\mathsf{types}(op(e_1, \dots, e_n)) &= \textstyle\bigcup_{1 \le i \le n} \mathsf{types}(e_i) \\
\mathsf{types}(\langle \{x{:}B \mid e_1\}, e_2, k\rangle^l) &= \mathsf{types}(\{x{:}B \mid e_1\}) \cup \mathsf{types}(e_2) \\
\mathsf{types}(\langle \{x{:}B \mid e_1\}, s, r, k, e\rangle^\bullet) &= \\
&\mathsf{types}(\{x{:}B \mid e_1\}) \cup \mathsf{types}(r) \cup \mathsf{types}(e) \\
\mathsf{types}(\Uparrow l) &= \emptyset
\end{aligned}$$

Type, type set, and coercion type extraction

$\boxed{\mathsf{types}(T) : \mathcal{P}(T)}$

$$\begin{aligned}
\mathsf{types}(\{x{:}B \mid e\}) &= \{\{x{:}B \mid e\}\} \cup \mathsf{types}(e) \\
\mathsf{types}(T_1 \to T_2) &= \{T_1 \to T_2\} \cup \\
&\quad\ \mathsf{types}(T_1) \cup \mathsf{types}(T_2)
\end{aligned}$$

$\boxed{\mathsf{types}(a) : \mathcal{P}(T)}$

$$\begin{aligned}
\mathsf{types}(\bullet) &= \emptyset \\
\mathsf{types}(\mathsf{nil}) &= \emptyset \\
\mathsf{types}(\{x{:}B \mid e\}^l, r) &= \{\{x{:}B \mid e\}\} \cup \mathsf{types}(r) \\
\mathsf{types}(c_1 \mapsto c_2) &= \mathsf{types}(c_1) \cup \mathsf{types}(c_2)
\end{aligned}$$

Type height $\boxed{\mathsf{height}(T)}$

$$\begin{aligned}
\mathsf{height}(\{x{:}B \mid e\}) &= 1 \\
\mathsf{height}(T_1 \to T_2) &= 1 + \max_{i \in \{1,2\}} \mathsf{height}(T_i)
\end{aligned}$$

Figure 7. Type extraction and type height

C. Proofs of bounds for space-efficiency

This section contains our definitions for collecting types in a program and the corresponding proof of bounded space consumption (for all modes at once).

We define a function collecting all of the distinct types that appear in a program in Figure 7. If the type $T = \{x{:}\mathsf{Int} \mid x \geq 0\} \to \{y{:}\mathsf{Int} \mid y \neq 0\}$ appears in the program e, then $\mathsf{types}(e)$ includes the type T itself along with its subparts $\{x{:}\mathsf{Int} \mid x \geq 0\}$ and $\{y{:}\mathsf{Int} \mid y \neq 0\}$.

C.1 Lemma: $\mathsf{types}(e[e'/x]) \subseteq \mathsf{types}(e) \cup \mathsf{types}(e')$

Proof: By induction on e. $\square$

C.2 Lemma: $\mathsf{types}(\mathsf{dom}(a)) \subseteq \mathsf{types}(a)$

Proof: This property is trivial when $a = \bullet$.
Immediate when $a = c_1 \mapsto c_2$. $\square$

C.3 Lemma: $\mathsf{types}(\mathsf{cod}(a)) \subseteq \mathsf{types}(a)$

Proof: Similar to Lemma C.2. $\square$

C.4 Lemma [Coercing types doesn't introduce types]: $\mathsf{types}(\mathsf{coerce}(T_1, T_2, l)) \subseteq \mathsf{types}(T_1) \cup \mathsf{types}(T_2)$

Proof: By induction on T_1 and T_2. When they are refinements, we have the coercion just being $\{x{:}B \mid e_2\}^l$. When they are functions, by the IH. $\square$

C.5 Lemma [Dropping types doesn't introduce types]: $\mathsf{types}(\mathsf{drop}\,(r, \{x{:}B \mid e\})) \subseteq \mathsf{types}(r)$

Proof: By induction on r.

($r = \mathsf{nil}$) The two sides are immediately equal.

($r = \{x{:}B \mid e'\}^l, r'$) If $\{x{:}B \mid e'\} \not\supset \{x{:}B \mid e\}$, then the two are identical. If not, then we have $\mathsf{types}(r') \subseteq \mathsf{types}(r)$ by the IH.

$\square$

C.6 Lemma [Coercion merges don't introduce types]: $\mathsf{types}(\mathsf{join}(r_1, r_2)) \subseteq \mathsf{types}(r_1) \cup \mathsf{types}(r_2)$

Proof: By induction on r_1.

($r_1 = \mathsf{nil}$) The two sides are immediately equal.

($r_1 = \{x{:}B \mid e\}^l, r_1'$) Using Lemma C.5, we find:

$$\begin{aligned}
\mathsf{types}(\mathsf{join}(r_1, r_2)) &= \{\{x{:}B \mid e\}\} \cup \\
&\quad\ \mathsf{types}(\mathsf{join}(r_1', \mathsf{drop}\,(r_2, \{x{:}B \mid e\}))) \\
&\subseteq \{\{x{:}B \mid e\}\} \cup \mathsf{types}(r_1') \cup \\
&\quad\ \mathsf{types}(\mathsf{drop}\,(r_2, \{x{:}B \mid e\})) \\
&\subseteq \{\{x{:}B \mid e\}\} \cup \mathsf{types}(r_1') \cup \mathsf{types}(r_2) \\
&= \mathsf{types}(r_1) \cup \mathsf{types}(r_2)
\end{aligned}$$

$\square$

C.7 Lemma [Reduction doesn't introduce types]: If $e \longrightarrow_m e'$ then $\mathsf{types}(e') \subseteq \mathsf{types}(e)$.

Proof: By induction on the step taken. $\square$