Space-Efficient Manifest Contracts

Anonymous

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. We define a framework for space efficiency, traversing the design space with three different space-efficient manifest calculi. Along the way, we examine the diverse correctness criteria for contract semantics; we conclude with a language whose contracts enjoy (galactically) bounded, *sound* space consumption—they are observationally equivalent to the standard, space-inefficient semantics.

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 programmers 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.

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 [6]. 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. Checking a function f against the contract $T_1 \rightarrow T_2$ is deferred: we record the contract on f by *wrapping* it with a *function proxy*. Calling this function proxy with an argument e, 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. Models of contract calculi have taken two forms: latent and manifest [11], and both suffer from space inefficiency. We take the manifest approach here, which means checking contracts with *casts*, written $\langle T_1 \Rightarrow T_2 \rangle^l\ e$. 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 [6]. 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:

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

Findler and Felleisen neatly designed a system for contract checking in a higher-order world, but there is a problem: casts are space inefficient [14].

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:

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

A cast insertion algorithm [26] might produce the following non-tail recursive function:

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

Tail-call optimization is essential for usable functional languages—we believe that 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* [24], a form of (manifest) contracts designed to mediate dynamic and simple typing—that is, gradual types (a) allows the dynamic type, and

(b) restricts the predicates in contracts to checks on type tags. Herman et al. [14] developed the first space-efficient gradually typed system, using Henglein's coercions [13]; Siek and Wadler [25] 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 offer three *modes* of space-efficiency; all of the modes are defined in a single calculus which we call λ_{H}. Each mode enjoys varying levels of soundness with respect to the standard, space-inefficient semantics of classic λ_{H}. We sketch here the mode-indexed rules for combining annotations on casts—the key rules for space efficiency.

The *forgetful* mode uses empty annotations, •; we combine two casts by dropping intermediate types:

$$\langle T_2 \overset{\bullet}{\Rightarrow} T_3 \rangle^{l_2}\ (\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^{l_1}\ e) \longrightarrow_{\mathsf{F}} \langle T_1 \overset{\bullet}{\Rightarrow} T_3 \rangle^{l_2}\ e$$

Surprisingly, this evaluation rule is type safe and somewhat sound with respect to the classic mode, as discovered by Greenberg [10]: if classic λ_{H} produces a value, so does forgetful λ_{H}.

The *heedful* mode uses sets of types as its annotations, making sure to save the intermediate type:

$$\langle T_2 \overset{\mathcal{S}_2}{\Rightarrow} T_3 \rangle^{l_2}\ (\langle T_1 \overset{\mathcal{S}_1}{\Rightarrow} T_2 \rangle^{l_1}\ e) \longrightarrow_{\mathsf{H}} \langle T_1 \overset{\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}}{\Rightarrow} T_3 \rangle^{l_2}\ e$$

In Siek and Wadler's terms, we use the powerset lattice for annotations, while pointed types. Heedful and classic λ_{H} are almost identical, except sometimes they blame different labels.

Finally, the *eidetic* mode annotates casts with *refinement lists* and *function coercions*—a new form of coercion inspired by Greenberg [10]. The coercions keep track of checking so well that the type indices and blame labels on casts are unnecessary:

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

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.

Since eidetic and classic λ_{H} behave the same, why bother with forgetful and heedful? First and foremost, the calculi offer insights into the semantics of contracts. Second, we offer them as alternative points in the design space. Finally and perhaps cynically, they are strawmen—warm up exercises for eidetic λ_{H}.

We claim two contributions:

1. 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 [10], 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.
2. A framework for defining space efficiency in manifest contract systems, with an exploration of the design space. We identify common structures and methods in the operational semantics as well as in the proofs of type soundness, soundness with regard to the classic framework, and space bounds.

We do not prove a blame theorem [27], since we lack the clear separation of dynamic and static typing found in gradual typing. We conjecture that such a theorem could be proved for classic and eidetic λ_{H}—but perhaps not for forgetful and heedful λ_{H}, which skip checks and change blame labels. 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 [18]. Our contribution is showing that sound space efficiency is *possible* where it was believed to be impossible [10]; we leave evidence that it is *practicable* for future work. Our proofs are available in the non-blind supplementary material, Appendices A–C.

Readers who are very familiar with this topic can read Figures 1, 2, and 3 and then skip directly to Section 3.5. Readers who understand the space inefficiency of contracts but not 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. PLT Racket's contract system [21], 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:

$$\begin{array}{rcl@{\quad}l}
\mathsf{null} & : & \alpha\ \mathsf{List} \to \{x{:}\mathsf{Bool} \mid \mathsf{true}\} & = \ldots \\
\mathsf{head} & : & \{x{:}\alpha\ \mathsf{List} \mid \mathsf{not}\ (\mathsf{null}\ x)\} \to \alpha & = \ldots \\
\\
\mathsf{empty} & : & \alpha\ \mathsf{Set} \to \{x{:}\mathsf{Bool} \mid \mathsf{true}\} & = \mathsf{null} \\
\mathsf{min} & : & \{x{:}\alpha\ \mathsf{Set} \mid \mathsf{not}\ (\mathsf{empty}\ x)\} \to \alpha & = \mathsf{head}
\end{array}$$

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 PLT Racket encourage modules to redeclare contracts to avoid being blamed—which can result in redundant checking.

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 square canvas dimensions, where $\mathsf{SquareCanvas} = \{x{:}\mathsf{Canvas} \mid \mathsf{width}(x) = \mathsf{height}(x)\}$:

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

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 func-

Modes

$$\begin{array}{rcll} m & ::= & \mathsf{C} & \text{\underline{c}lassic } \lambda_{\mathrm{H}}\text{; Section 3} \\ & | & \mathsf{F} & \text{\underline{f}orgetful } \lambda_{\mathrm{H}}\text{; Section 4} \\ & | & \mathsf{H} & \text{\underline{h}eedful } \lambda_{\mathrm{H}}\text{; Section 5} \\ & | & \mathsf{E} & \text{\underline{e}idetic } \lambda_{\mathrm{H}}\text{; Section 6} \end{array}$$

Types

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

Terms

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

Annotations: type set, coercions, and refinement lists

$$\begin{array}{rcl} a & ::= & \bullet \mid \mathcal{S} \mid c \\ \mathcal{S} & ::= & \emptyset \mid \{T_1, \ldots, T_n\} \\ c & ::= & r \mid c_1 \mapsto c_2 \\ r & ::= & \mathsf{nil} \mid \{x{:}B \mid e\}^l, r \end{array}$$

Statuses

$$s ::= \checkmark \mid ?$$

Locations

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

Figure 1. Syntax of λ_{H}

tions *back* to the refined types. Calling flipH (flipH p) yields:

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

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 that the underlying implementation of primFlipH is doing. A space-efficient scheme for manifest contracts bounds the number of function proxies that can accumulate. 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.

3. Classic manifest contracts

The standard manifest contract calculus, λ_{H}, is originally due to Flanagan [8]. 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 four modes of λ_{H}: classic λ_{H}, mode C; forgetful λ_{H}, mode F; heedful λ_{H}, mode H; and eidetic λ_{H}, mode E. Each of these languages uses the syntax of Figure 1, while the typing rules and operational semantics are indexed by the mode m. We summarize how each of these modes differ in Section 3.2—but first we (laconically) explain the syntax and the interesting bits of the operational semantics.

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 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 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 and forgetful λ_{H} don't need annotations—we write $\bullet$. Heedful λ_{H} uses type sets $\mathcal{S}$ to track space-efficiently pending checks. Eidetic λ_{H} uses coercions c.

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 in 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 6.

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.

Each mode defines its own value rule for function proxies, V_PROXYm. 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. Other modes only allow lambda abstractions to be proxied. All of the space-efficient calculi take our approach, where a function cast applied to a value *is* a value; some space inefficient ones do, too [6, 11, 12]. In some 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 [2, 6, 8, 24]. 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. While we can imagine a third semantics that looks into closures rather than having explicit function proxies, we decline to gaze long into the abyss of lambda abstractions, lest they also gaze back into us.

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 3.3). 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 said that "well typed expressions don't go wrong" [19]; 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"—that supports our forgetful semantics (Section 4).

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. The E_CHECK* rules manage active checks in the conventional way; heedful and eidetic use slightly different forms, described in their respective sections.

Since space bounds rely not only on limiting the number of function proxies but also accumulation of casts on the stack, the

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

$$\frac{}{\mathsf{val}_m\ k}\ \text{V\_CONST} \qquad \frac{}{\mathsf{val}_m\ \lambda x{:}T.\ e}\ \text{V\_ABS} \qquad \frac{\mathsf{val}_{\mathsf{C}}\ e}{\mathsf{val}_{\mathsf{C}}\ \langle T_{11}\to T_{12}\overset{\bullet}{\Rrightarrow} T_{21}\to T_{22}\rangle^l\ e}\ \text{V\_PROXYC} \qquad \frac{\mathsf{val}_m\ e}{\mathsf{result}_m\ e}\ \text{R\_VAL} \qquad \frac{}{\mathsf{result}_m\ \Uparrow l}\ \text{R\_BLAME}$$

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

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

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

$$\begin{array}{rcl} \mathsf{dom}(\bullet) & = & \bullet \\ \mathsf{dom}(\mathcal{S}) & = & \bigcup_{T\in\mathcal{S}} \mathsf{dom}(T) \\ \mathsf{dom}(c_1 \mapsto c_2) & = & c_1 \end{array} \qquad \begin{array}{rcl} \mathsf{cod}(\bullet) & = & \bullet \\ \mathsf{cod}(\mathcal{S}) & = & \bigcup_{T\in\mathcal{S}} \mathsf{cod}(T) \\ \mathsf{cod}(c_1 \mapsto c_2) & = & c_2 \end{array}$$

$$\frac{m\ \in\ \{\mathsf{C},\mathsf{F}\}}{\langle\{x{:}B \mid e_1\}\overset{\bullet}{\Rrightarrow}\{x{:}B \mid e_2\}\rangle^l\ k\ \longrightarrow_m\ \langle\{x{:}B \mid e_2\}, e_2[k/x], k\rangle^l}\ \text{E\_CHECKNONE}$$

$$\frac{}{\langle\{x{:}B \mid e\}, \mathsf{true}, k\rangle^l\ \longrightarrow_m\ k}\ \text{E\_CHECKOK} \qquad \frac{}{\langle\{x{:}B \mid e\}, \mathsf{false}, k\rangle^l\ \longrightarrow_m\ \Uparrow l}\ \text{E\_CHECKFAIL}$$

$$\frac{e_1\ \longrightarrow_m\ e_1'}{e_1\ e_2\ \longrightarrow_m\ e_1'\ e_2}\ \text{E\_APPL} \qquad \frac{\mathsf{val}_m\ e_1 \quad e_2\ \longrightarrow_m\ e_2'}{e_1\ e_2\ \longrightarrow_m\ e_1\ e_2'}\ \text{E\_APPR}$$

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

$$\frac{e\ \longrightarrow_{\mathsf{C}}\ e'}{\langle T_1\overset{\bullet}{\Rrightarrow} T_2\rangle^l\ e\ \longrightarrow_{\mathsf{C}}\ \langle T_1\overset{\bullet}{\Rrightarrow} T_2\rangle^l\ e'}\ \text{E\_CASTINNERC} \qquad \frac{e_2\ \longrightarrow_m\ e_2'}{\langle\{x{:}B \mid e_1\}, e_2, k\rangle^l\ \longrightarrow_m\ \langle\{x{:}B \mid e_1\}, e_2', k\rangle^l}\ \text{E\_CHECKINNER}$$

$$\frac{m \neq \mathsf{C} \quad e_2\ \longrightarrow_m\ e_2' \quad e_2 \neq \langle T_1\overset{a'}{\Rrightarrow} T_2\rangle^{l'}\ e_2''}{\langle T_2\overset{a}{\Rrightarrow} T_3\rangle^l\ e_2\ \longrightarrow_m\ \langle T_2\overset{a}{\Rrightarrow} T_3\rangle^l\ e_2'}\ \text{E\_CASTINNER} \qquad \frac{a_3 = \mathsf{merge}_m(T_1, a_1, T_2, a_2, T_3)}{\langle T_2\overset{a_2}{\Rrightarrow} T_3\rangle^l\ (\langle T_1\overset{a_1}{\Rrightarrow} T_2\rangle^{l'}\ e_2)\ \longrightarrow_m\ \langle T_1\overset{a_3}{\Rrightarrow} T_3\rangle^l\ e_2}\ \text{E\_CASTMERGE}$$

$$\frac{}{\Uparrow l\ e_2\ \longrightarrow_m\ \Uparrow l}\ \text{E\_APPRAISEL} \qquad \frac{\mathsf{val}_m\ e_1}{e_1\ \Uparrow l\ \longrightarrow_m\ \Uparrow l}\ \text{E\_APPRAISER} \qquad \frac{}{\langle T_1\overset{\mathcal{S}}{\Rrightarrow} T_2\rangle^l\ \Uparrow l'\ \longrightarrow_m\ \Uparrow l'}\ \text{E\_CASTRAISE}$$

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

Figure 2. Core operational semantics of $\lambda_{\rm H}$; classic $\lambda_{\rm H}$ rules are salmon; space-efficient rules are periwinkle

core semantics doesn't include a cast congruence rule. The congruence rule for casts in classic $\lambda_{\rm H}$, E_CASTINNERC, allows for free use of congruence. In the space-efficient calculi, the use of congruence is instead limited by the rules E_CASTINNER and E_CASTMERGE. Cast arguments only take congruent steps when they aren't casts themselves. A cast applied to another cast *merges*, using the merge function. Each space-efficient calculus uses a different annotation scheme, so each one has a different merge function. We deliberately leave merge undefined sometimes—heedful and eidetic $\lambda_{\rm H}$ must control when E_CASTMERGE can apply. Note that we don't need to specify $m \neq \mathsf{C}$ in E_CASTMERGE—we just don't define a merge operator for classic $\lambda_{\rm H}$. We have E_CASTMERGE arbitrarily retain the label of the outer cast. No choice is "right" here—we discuss this issue further in Section 5.

3.2 Cast merges by example

Each mode's section explains its semantics in detail, but we can summarize the cast merging rules here by example. Consider the following term:

$$\begin{aligned} e = {} & \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}\overset{\bullet}{\Rrightarrow}\{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\ & (\langle\{x{:}\mathsf{Int} \mid x \geq 0\}\overset{\bullet}{\Rrightarrow}\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}\rangle^{l_2} \\ & \quad (\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\}\overset{\bullet}{\Rrightarrow}\{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^{l_1}\ {-1})) \end{aligned}$$

Here e runs three checks on integer -1: first for non-negativity (blaming l_1 on failure), then for evenness (blaming l_2 on failure), and then for non-zeroness (blaming l_3 on failure). Classic and eidetic $\lambda_{\rm H}$ both blame l_1; heedful $\lambda_{\rm H}$ also raises blame, though it blames a different label, l_3; forgetful $\lambda_{\rm H}$ actually *accepts* the value, returning -1. We discuss the operational rules for modes other than C in detail in each mode's section; for now, we repeat the derived rules for merging casts from Section 1.

Classic $\lambda_{\rm H}$ evaluates the casts step-by-step: first it checks whether -1 is positive, which fails, so $e\ \longrightarrow_{\mathsf{C}}^{*}\ \Uparrow l_1$. Forgetful $\lambda_{\rm H}$ doesn't use annotations at all—it just forgets the intermediate casts, effectively using the following rule:

$$\langle T_2\overset{\bullet}{\Rrightarrow} T_3\rangle^{l_2}\ (\langle T_1\overset{\bullet}{\Rrightarrow} T_2\rangle^{l_1}\ e)\ \longrightarrow_{\mathsf{F}}\ \langle T_1\overset{\bullet}{\Rrightarrow} T_3\rangle^{l_2}\ e$$

It never checks for non-negativity or evenness, skipping straight to the check that -1 is non-zero. Heedful λ_{H} works by annotating casts with a set of intermediate types, effectively using the rule:

$$\langle T_2 \overset{\mathcal{S}_2}{\Rightarrow} T_3\rangle^{l_2} \; (\langle T_1 \overset{\mathcal{S}_1}{\Rightarrow} T_2\rangle^{l_1} \; e) \longrightarrow_{\mathsf{H}} \langle T_1 \overset{\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}}{\Rightarrow} T_3\rangle^{l_2} \; e$$

Every type in a type set needs to be checked, but the order is essentially nondeterministic: heedful λ_{H} checks that -1 is non-negative and even in *some* order. Whichever one is checked first fails; both cases raise $\Uparrow l_3$. Finally, eidetic λ_{H} uses coercions as its annotations; coercions c are detailed checking plans for running checks in the same order as classic λ_{H} while skipping redundant checks. As we will see in Section 6, eidetic λ_{H} generates coercions and then drops blame labels, giving us the rule:

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

There are no redundant checks in the example term e, so eidetic λ_{H} does *exactly* the same checking as classic, finding $e \longrightarrow_{\mathsf{E}}^{*} \Uparrow l_1$.

3.3 Type system

All three 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 8.

Context well formedness is entirely straightforward; type well formedness requires some care to get base types off the ground. 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 $T_1 \rightarrow T_2$. Every cast must be between compatible types; at their core, λ_{H} programs are well typed simply typed lambda calculus programs.

Our family of calculi use different annotations. All source programs (defined below) begin without annotations—we write the empty annotation $\bullet$. 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 $\vdash T_1 \parallel T_2$.

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 [8, 11, 16, 17] and the extra rule of others [2]. 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 $\vdash_m \{x{:}B_1 \mid e_1\} \rightarrow \ldots \rightarrow \{x{:}B_n \mid e_n\}$. 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—because forgetful λ_{H} may skip them, leading to different typings. We believe this is not so stringent a requirement: the types for operations ought to be simple, e.g. $\mathsf{ty}(\mathsf{div}) = \{x{:}\mathsf{Real} \mid \mathsf{true}\} \rightarrow \{y{:}\mathsf{Real} \mid y \neq 0\} \rightarrow \{z{:}\mathsf{Real} \mid \mathsf{true}\}$, 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 [3, 8, 17]; 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.

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 (Appendix A).

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

- T_CONST only ever assigns the type $\{x{:}\mathsf{ty}(k) \mid \mathsf{true}\}$.
- Casts have empty annotations $a = \bullet$.
- T_CHECK, T_STACK (Section 6), and T_BLAME are not used.

3.4 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 [28]. But other standard metatheoretical machinery—weakening, substitution, and regularity—can be proved for all modes at once (see Section A.1 in the technical appendix). To wit, we prove syntactic type soundness in Section A.2 for classic λ_{H} in just three mode-specific lemmas: canonical forms, progress, and preservation.

3.5 Overview

In the rest of this paper, we give the semantics for three space-efficient modes for λ_{H}, relating the languages' behavior on source programs (Definition 3.1). The forgetful mode is space efficient without annotations, converging to a value more often than classic λ_{H} ($m = \mathsf{F}$; Section 4). The heedful mode is space efficient and uses type sets to converge to a value exactly when classic λ_{H} does; it may blame different labels, though ($m = \mathsf{H}$; Section 5). The eidetic mode is space efficient and uses coercions to track pending checks; it behaves exactly like classic λ_{H} ($m = \mathsf{E}$; Section 6).

One may wonder why we even bother to mention forgetful and heedful λ_{H}, if eidetic λ_{H} is soundly space efficient with respect to classic λ_{H}. These two 'intermediate' modes are interesting as an exploration of the design space—but also in their own right.

Forgetful λ_{H} takes a radical approach that involves skipping checks—its soundness is rather surprising and offers insights into the semantics of contracts. Contracts have been used for more than avoiding wrongness, though: they have been used in PLT Racket for abstraction and information hiding [20, 21]. Forgetful λ_{H} can't use contracts for information hiding. Suppose we implement user records as functions from strings to strings. We would like to pass a user record to an untrusted component, hiding some fields but not others. We can achieve this by specifying a white- or blacklist in a contract, e.g., $\{f{:}\mathsf{String} \mid f \neq \text{“}\texttt{password}\text{”}\} \rightarrow \{v{:}\mathsf{String} \mid \mathsf{true}\}$. Wrapping a function in this contract introduces a function proxy... which can be overwritten by E_CASTMERGE! To really get information hiding, the programmer must explicitly η-expand the

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

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

$$\frac{}{\vdash_m \{x{:}B \mid \mathsf{true}\}}\ \textsc{WF\_Base} \qquad \frac{x{:}\{x{:}B \mid \mathsf{true}\} \vdash_m e : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\vdash_m \{x{:}B \mid e\}}\ \textsc{WF\_Refine} \qquad \frac{\vdash_m T_1 \quad \vdash_m T_2}{\vdash_m T_1 \to T_2}\ \textsc{WF\_Fun}$$

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

$$\frac{}{\vdash \{x{:}B \mid e_1\} \parallel \{x{:}B \mid e_2\}}\ \textsc{S\_Refine} \qquad \frac{\vdash T_{11} \parallel T_{21} \quad \vdash T_{12} \parallel T_{22}}{\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}}\ \textsc{S\_Fun} \qquad \frac{\vdash T_1 \parallel T_2 \quad \vdash_m T_1 \quad \vdash_m T_2}{\vdash_m \bullet \parallel T_1 \Rightarrow T_2}\ \textsc{A\_None}$$

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

$$\frac{\vdash_m \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash_m x : T}\ \textsc{T\_Var} \qquad \frac{\vdash_m T_1 \quad \Gamma, x{:}T_1 \vdash_m e_{12} : T_2}{\Gamma \vdash_m \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \textsc{T\_Abs} \qquad \frac{\vdash_m \Gamma \quad \vdash_m T}{\Gamma \vdash_m \Uparrow l : T}\ \textsc{T\_Blame}$$

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

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

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

Figure 3. Universal typing rules of λ_{H}

function proxy, writing $(\lambda f{:}\{f{:}\mathsf{String} \mid f \neq \text{“}\mathtt{password}\text{''}\}.\ \ldots)$. Forgetful λ_{H}'s contracts can't enforce abstractions.

While Siek and Wadler [25] uses the lattice of type precision in their threesomes without blame, our heedful λ_{H} uses the powerset lattice of types. Just as Siek and Wadler use labeled types and meet-like composition for threesomes *with* blame, we may be able to derive something similar for heedful and eidetic λ_{H}: in a (non-commutative) skew lattice, heedful uses a potentially re-ordering conjunction while eidetic preserves order. A lattice-theoretic account of casts, coercions, and blame may be possible.

4. Forgetful space efficiency

In forgetful λ_{H}, we offer a simple solution to space-inefficient casts: just forget about them. Function proxies only ever wrap lambda abstractions; trying to cast a function proxy simply throws away the inner proxy. Just the same, when accumulating casts on the stack, we throw away all but the last cast. Readers may wonder: how can this ever be sound? Several factors work together to make forgetful λ_{H} a sound calculus. In short, the key ingredients are call-by-value evaluation and the observation that (one understanding of) type safety only talks about reduction to values.

In this section, our mode $m = \mathsf{F}$: our evaluation relation is $\longrightarrow_{\mathsf{F}}$ and we use typing judgments of the form, e.g. $\Gamma \vdash_{\mathsf{F}} e : T$. Forgetful λ_{H} is the simplest of the space-efficient calculi: it just uses the standard typing rules from Figure 3 and the space-efficient reduction rules from Figure 2. We give the new operational definitions for $m = \mathsf{F}$ in Figure 4: a new value rule and the definition of the merge operator. First, V_ProxyF says that function proxies in forgetful λ_{H} are only values when the proxied value is a lambda (and not another function proxy). Limiting the number of function proxies is critical for establishing a space bounds, as we do in Section 8. Forgetful casts don't use annotations, so they just use A_None. The forgetful merge operator just *forgets* the intermediate type T_2.

We demonstrate this semantics on the example from Section 3.1 in Figure 5. We first step by merging casts, forgetting the intermedi-

Values and merging $\boxed{\mathsf{val}_{\mathsf{F}}\ e}$

$$\frac{}{\mathsf{val}_{\mathsf{F}}\ \langle T_{11} \to T_{12} \overset{\emptyset}{\Rightarrow} T_{21} \to T_{22} \rangle^l\ \lambda x{:}T.\ e}\ \textsc{V\_ProxyF}$$

$$\mathsf{merge}_{\mathsf{F}}(T_1, \bullet, T_2, \bullet, T_3) = \bullet$$

Figure 4. Operational semantics of forgetful λ_{H}

$$\begin{array}{l}
e = \langle \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \rangle^{l_2} \\
\quad\quad (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1})) \\
\hfill (\textsc{E\_CastMerge}) \\
\longrightarrow_{\mathsf{F}} \langle \{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3} \\
\quad (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\} \rangle^{l_1}\ {-1}) \\
\hfill (\textsc{E\_CastMerge}) \\
\longrightarrow_{\mathsf{F}} \langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\} \rangle^{l_3}\ {-1} \\
\hfill (\textsc{E\_CheckNone}) \\
\longrightarrow_{\mathsf{F}} \langle \{x{:}\mathsf{Int} \mid x \neq 0\}, -1 \neq 0, -1 \rangle^{l_3} \\
\hfill (\textsc{E\_CheckInner}/\textsc{E\_Op}) \\
\longrightarrow_{\mathsf{F}} \langle \{x{:}\mathsf{Int} \mid x \neq 0\}, \mathsf{true}, -1 \rangle^{l_3} \qquad (\textsc{E\_CheckOK}) \\
\longrightarrow_{\mathsf{F}} -1
\end{array}$$

Figure 5. Example of forgetful λ_{H}

ate type. Then contract checking proceeds as normal for the target type; since -1 is non-zero, the check succeeds and returns its scrutinee by E_CHECKOK.

The type soundness property typically has two parts: (a) well typed programs don't go 'wrong' (for us, getting stuck), and (b) well typed programs reduce to programs that are well typed at the same type. How could a forgetful λ_{H} program go wrong, violating property (a)? The general "skeletal" structure of types means we never have to worry about errors caught by simple type systems, such as trying to apply a non-function. Our semantics can get stuck by trying to apply an operator to an input that isn't in its domain, e.g., trying to divide by zero. To guarantee that we avoid stuck operators, λ_{H} generally relies on subject reduction, property (b). Operators are assigned types that avoid stuckness, i.e., $\mathsf{ty}(op)$ and $[\![op]\!]$ agree. Some earlier systems have done this [8, 16] while others haven't [2, 11]. We view it as a critical component of contract calculi. So for, say, integer division, $\mathsf{ty}(\mathsf{div}) = \{x{:}\mathsf{Int} \mid \mathsf{true}\} \rightarrow \{y{:}\mathsf{Int} \mid y \neq 0\} \rightarrow \{z{:}\mathsf{Int} \mid \mathsf{true}\}$. To actually use div in a program, the second argument must be typed as a non-zero integer—by a non-source typing with T_CONST directly (see Definition 3.1) or by casting (T_CAST). It may seem dangerous: casts protect operators from improper values, preventing stuckness; forgetful λ_{H} eliminates some casts. But consider the cast eliminated by E_CASTMERGE:

$$\langle T_2 \overset{\bullet}{\Rightarrow} T_3 \rangle^{l} \; (\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^{l'} \; e) \; \longrightarrow_{\mathsf{F}} \; \langle T_1 \overset{\bullet}{\Rightarrow} T_3 \rangle^{l} \; e$$

While the program tried to cast e to a T_2, it immediately cast it back out—no operation relies on e being a T_2. Skipping the check doesn't risk stuckness. Since λ_{H} is call-by-value, we can use the same reasoning to allow functions to assume that their inputs inhabit their types—a critical property for programmer reasoning.

Forgetful λ_{H} enjoys soundness via a standard syntactic proof, reusing the theorems from Section A.1. What's more, source programs are well typed in classic λ_{H} iff they are well typed in forgetful λ_{H}: both languages can run the same terms. Proofs are in Section A.3.

5. Heedful space efficiency

Heedful λ_{H} ($m = \mathsf{H}$) takes the cast merging strategy from forgetful λ_{H}, but uses *type sets* on casts and function proxies to avoid dropping casts. Space efficiency for heedful λ_{H} rests on the use of sets: classic λ_{H} allows for arbitrary lists of function proxies and casts on the stack to accumulate. Restricting this accumulation to a set gives us a straightforward bound on the amount of accumulation: a program of fixed size can only have so many types at each size. We discuss this idea further in Section 8.

We extend the typing rules and operational semantics Figure 6. Up until this point, we haven't used annotations. Heedful λ_{H} collects type sets as casts merge to record the types that must be checked. The A_TYPESET annotation well formedness rule extends the premises of A_NONE with the requirement that if $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_1 \Rightarrow T_2$, then all the types in $\mathcal{S}$ are well formed and compatible with T_1 and T_2. Type set compatibility is stable under removing elements from the set $\mathcal{S}$, and it is symmetric and transitive with respective to its type indices (since compatibility itself is symmetric and transitive).

One might expect the types in type sets to carry blame labels—might we then be able to have *sound* space efficiency? It turns out that just having labels in the sets isn't enough—we actually need to keep track of the ordering of checks. Eidetic λ_{H} (Section 6) does exactly this tracking. Consider this calculus a warmup.

Heedful λ_{H} adds some evaluation rules to the universal ones found in Figure 2. First E_TYPESET takes a source program cast without an annotation and annotates it with an empty set.

Type set well formedness $\quad \boxed{\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2}$

$$\frac{\vdash T_1 \parallel T_2 \quad \vdash_{\mathsf{H}} T_1 \quad \vdash_{\mathsf{H}} T_2 \qquad \forall T \in \mathcal{S}.\, \vdash_{\mathsf{H}} T \quad \vdash T \parallel T_1}{\vdash_{\mathsf{H}} \mathcal{S} \parallel T_1 \Rightarrow T_2} \;\text{A\_TYPESET}$$

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

$$\frac{}{\mathsf{val}_{\mathsf{H}}\; \langle T_{11} \rightarrow T_{12} \overset{\mathcal{S}}{\Rightarrow} T_{21} \rightarrow T_{22} \rangle^{l}\; \lambda x{:}T.\; e} \;\text{V\_PROXYH}$$

$$\frac{}{\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^{l}\; e \longrightarrow_{\mathsf{H}} \langle T_1 \overset{\emptyset}{\Rightarrow} T_2 \rangle^{l}\; e} \;\text{E\_TYPESET}$$

$$\frac{}{\begin{array}{l}\langle \{x{:}B \mid e_1\} \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_2\} \rangle^{l}\; k \longrightarrow_{\mathsf{H}} \\ \langle \{x{:}B \mid e_2\}, e_2[k/x], k \rangle^{l}\end{array}} \;\text{E\_CHECKEMPTY}$$

$$\frac{\mathsf{choose}(\mathcal{S}) = \{x{:}B \mid e_2\}}{\begin{array}{l}\langle \{x{:}B \mid e_1\} \overset{\mathcal{S}}{\Rightarrow} \{x{:}B \mid e_3\} \rangle^{l}\; k \longrightarrow_{\mathsf{H}} \\ \langle \{x{:}B \mid e_2\} \overset{\mathcal{S} \setminus \{x:B \mid e_2\}}{\Rightarrow} \{x{:}B \mid e_3\} \rangle^{l} \\ \quad \langle \{x{:}B \mid e_2\}, e_2[k/x], k \rangle^{l}\end{array}} \;\text{E\_CHECKSET}$$

$$\mathsf{choose}(\mathcal{S}) \in \mathcal{S} \text{ when } \mathcal{S} \neq \emptyset$$

$$\mathsf{merge}_{\mathsf{H}}(T_1, \mathcal{S}_1, T_2, \mathcal{S}_2, T_3) = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}$$

$$\begin{array}{rcl}\mathsf{dom}(\mathcal{S}) & = & \bigcup_{T \in \mathcal{S}} \mathsf{dom}(T) \\ \mathsf{cod}(\mathcal{S}) & = & \bigcup_{T \in \mathcal{S}} \mathsf{cod}(T)\end{array}$$

Figure 6. Annotation typing and operational semantics of heedful λ_{H}

E_CHECKEMPTY is exactly like E_CHECKNONE, though we separate the two to avoid conflating the empty annotation $\bullet$ and the empty set $\emptyset$. In E_CHECKSET, we use an essentially unspecified function choose to pick a type from a type set to check. Using the choose function is theoretically expedient, as it hides all of heedful λ_{H}'s nondeterminism. Nothing is inherently problematic with this nondeterminism, but putting it in the reduction relation itself complicates the proof of strong normalization that is necessary for the proof relating classic and heedful λ_{H} (Section 7).

For function types, we define $\mathsf{dom}(\mathcal{S})$ and $\mathsf{cod}(\mathcal{S})$ by mapping the underlying function on types over the set. Note that this may shrink the size of the set $\mathcal{S}$, but never grow it—there can't be more unique (co)domain types in $\mathcal{S}$ than there are types.

The merge operator, used in E_CASTMERGE, merges two sets by unioning the two type sets with the intermediate type, i.e.:

$$\langle T_2 \overset{\mathcal{S}_2}{\Rightarrow} T_3 \rangle^{l_2} \; (\langle T_1 \overset{\mathcal{S}_1}{\Rightarrow} T_2 \rangle^{l_1} \; e) \; \longrightarrow_{\mathsf{H}} \; \langle T_1 \overset{\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}}{\Rightarrow} T_3 \rangle^{l_2} \; e$$

There are some subtle interactions here between the different annotations: we won't merge casts that haven't yet stepped by E_TYPESET because $\mathsf{merge}_{\mathsf{H}}(T_1, \bullet, T_2, \bullet, T_3)$ isn't defined.

We demonstrate the heedful semantics by returning to the example from Section 3.1 in Figure 7. To highlight the difference between classic and heedful λ_{H}, we select a choose function that has heedful check the refinements out of order, failing on the check for evenness rather than the check for positivity. The real source of difference, however, is that E_CASTMERGE takes the second blame label of the two casts it merges. Taking the first wouldn't be right, either: suppose that the target type of the l_1 cast wasn't $\{x{:}\mathsf{Int} \mid x \geq 0\}$, but some other type that -1 inhabits. Then classic λ_{H} would blame l_2, but heedful λ_{H} would have held onto l_1. The

$$
\begin{aligned}
&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\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\_TypeSet}, \text{E\_CastInner/E\_TypeSet}) \\
&\longrightarrow^{*}_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\emptyset}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\
&\quad (\langle\{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\emptyset}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}\rangle^{l_2} \\
&\quad\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\_CastMerge}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\{\{x:\mathsf{Int} \mid x \bmod 2 = 0\}\}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\
&\quad\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\_TypeSet}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\{\{x:\mathsf{Int} \mid x \bmod 2 = 0\}\}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\
&\quad\quad (\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\emptyset}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^{l_1}\ {-1}) \\
&\hfill (\text{E\_CastMerge}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\mathcal{S}}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3}\ {-1} \\
&\quad \text{where } \mathcal{S} = \{\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}, \{x{:}\mathsf{Int} \mid x \geq 0\}\} \\
&\quad (\text{E\_CheckSet}, \mathsf{choose}(\mathcal{S}) = \{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\mathcal{S}'}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\
&\quad \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}, -1 \bmod 2 = 0, -1\rangle^{l_3} \\
&\hfill \text{where } \mathcal{S}' = \{\{x{:}\mathsf{Int} \mid x \geq 0\}\} \\
&\hfill (\text{E\_CastInner/E\_CheckInner/E\_Op}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\mathcal{S}'}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\
&\quad \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}, 1 = 0, -1\rangle^{l_3} \\
&\hfill (\text{E\_CastInner/E\_CheckInner/E\_Op}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\} \overset{\mathcal{S}'}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3} \\
&\quad \langle\{x{:}\mathsf{Int} \mid x \bmod 2 = 0\}, \mathsf{false}, -1\rangle^{l_3} \\
&\hfill (\text{E\_CastInner/E\_CheckFail}) \\
&\longrightarrow_{\mathsf{H}} \langle\{x{:}\mathsf{Int} \mid x \geq 0\} \overset{\emptyset}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \neq 0\}\rangle^{l_3}\ {\Uparrow} l_3 \\
&\hfill (\text{E\_CastRaise}) \\
&\longrightarrow_{\mathsf{H}} {\Uparrow} l_3
\end{aligned}
$$

Figure 7. Example of heedful λ_H

solution to this blame tracking problem is to hold onto blame labels in annotations—which, again, is exactly what we do in eidetic λ_H.

The syntactic proof of type soundness for heedful λ_H appears in an appendix in Section A.4. We also have "source typing" for heedful λ_H: source programs are well typed when $m = \mathsf{C}$ iff they are well typed when $m = \mathsf{H}$. As a corollary, source programs are well typed in F if and only if they are well typed in H.

6. Eidetic space efficiency

Eidetic λ_H uses *coercions*, a more refined system of annotations than heedful λ_H's type sets. Coercions do two things that type sets don't: they retain check order, and they track blame. Our coercions are ultimately inspired by those of Henglein [13]; we discuss the relation in Section 9. Recall the syntax of coercions from Figure 1:

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

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 define the coercion well formedness rules, an additional typing rule, and reduction rules for eidetic λ_H in Figure 8. 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 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_CastMerge 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 8). 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 ✓ or ?: when the status is ✓, we or 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 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 differentiates what our requirement is: when $s = ?$, the target type is in r. When $s = ✓$, 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 must define a merge operator, $\mathsf{merge}_{\mathsf{E}}$. We define it in terms of the $\triangleright$ operator, which is very nearly concatenation on refinement lists and a contravariant homomorphism on function coercions—except that it refuses to allow duplicate types to appear, choosing the leftmost blame label. Contravariance means that $c_1 \triangleright c_2$ takes leftmost labels in positive positions and rightmost labels in negative ones. As we show below, this corresponds to the positive parts taking older labels and negative parts taking newer ones.

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

$$\frac{\begin{array}{c}\vdash_{\mathsf{E}} \{x{:}B \mid e_1\} \quad \vdash_{\mathsf{E}} \{x{:}B \mid e_2\} \quad \vdash_{\mathsf{E}} r \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\} \\ \forall \{x{:}B \mid e\} \in r.\, \vdash_{\mathsf{E}} \{x{:}B \mid e\} \quad \text{no duplicates in } r \quad \{x{:}B \mid e_2\} \in r\end{array}}{\vdash_{\mathsf{E}} r \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}} \ \text{A\_REFINE} \qquad \frac{\vdash_{\mathsf{E}} c_1 \parallel T_{21} \Rightarrow T_{11} \quad \vdash_{\mathsf{E}} c_2 \parallel T_{12} \Rightarrow T_{22}}{\vdash_{\mathsf{E}} c_1 \mapsto c_2 \parallel (T_{11} \to T_{12}) \Rightarrow (T_{21} \to T_{22})} \ \text{A\_FUN}$$

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

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

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

$$\frac{}{\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e \longrightarrow_{\mathsf{E}} \langle T_1 \overset{\mathsf{coerce}(T_1, T_2, l)}{\Rightarrow} T_2\rangle^\bullet\ e} \ \text{E\_COERCE} \qquad \frac{}{\langle \{x{:}B \mid e_1\} \overset{r}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^\bullet\ k \longrightarrow_{\mathsf{E}} \langle \{x{:}B \mid e_2\}, ?, r, k, k\rangle^\bullet} \ \text{E\_COERCESTACK}$$

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

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

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

Cast translation and coercion operations

$$\begin{array}{rcl} \mathsf{merge}_{\mathsf{E}}(T_1, c_1, T_2, c_2, T_3) & = & c_1 \rhd c_2 \\ \mathsf{dom}(c_1 \mapsto c_2) & = & c_1 \\ \mathsf{cod}(c_1 \mapsto c_2) & = & c_2 \end{array} \qquad \begin{array}{rcl} \mathsf{coerce}(\{x{:}B \mid e_1\}, \{x{:}B \mid e_2\}, l) & = & \{x{:}B \mid e_2\}^l \\ \mathsf{coerce}(T_{11} \to T_{12}, T_{21} \to T_{22}, l) & = & \mathsf{coerce}(T_{21}, T_{11}, l) \mapsto \mathsf{coerce}(T_{12}, T_{22}, l) \end{array}$$

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

$$\begin{array}{rcl} (\mathsf{nil} \setminus \{x{:}B \mid e\}) & = & \mathsf{nil} \\ (\{x{:}B \mid e_1\}^l, r \setminus \{x{:}B \mid e\}) & = & \begin{cases} r & e = e_1 \\ \{x{:}B \mid e_1\}^l, (r \setminus \{x{:}B \mid e\}) & e_1 \neq e \end{cases} \end{array} \qquad \begin{array}{rcl} \checkmark \vee (e_1 = e_2) & = & \checkmark \\ ? \vee (e_1 = e_2) & = & \begin{cases} \checkmark & e_1 = e_2 \\ ? & \text{otherwise} \end{cases} \end{array}$$

Figure 8. Typing rules and operational semantics for eidetic λ_{H}

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 $T_2 = \{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{x{:}\mathsf{Int} \mid x > 0\}$. Unfolding $(\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ v_1)\ v_2$ in classic λ_{H}, we see 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 [5, 10, 12, 14, 27]. Unlike casts, coercions directly express the sequence of checks to be performed. Consider the coercion generated from the cast above:

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

In this example, there is only a single blame label, l. Tracking blame labels is critical for exactly matching classic λ_{H}'s behavior. To see why, we return to our example from before in Figure 9. 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 example, 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:

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

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.

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

$$\begin{aligned}
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} \\
& \qquad (\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} \\
& \qquad (\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} \\
& \qquad (\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \overset{\bullet}{\Rightarrow} \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^{l_1}\ {-1})) \\
& \hfill (\text{E\_CASTMERGE}) \\
\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\_CASTMERGE}) \\
\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{aligned}$$

Figure 9. Example of eidetic λ_{H}

Value rules $\boxed{e_1 \sim_m e_2 : T}$

$$\begin{aligned}
k \sim_m k : \{x{:}B \mid e\} &\iff \mathsf{ty}(k) = B \wedge e[k/x] \longrightarrow_m^* \mathsf{true} \\
e_{11} \sim_m e_{21} : T_1 \to T_2 &\iff \mathsf{val}_{\mathsf{C}}\ e_1 \wedge \mathsf{val}_m\ e_2 \wedge \\
& \qquad \forall e_{12} \sim_m e_{22} : T_1.\ e_{11}\ e_{12} \simeq_m e_{21}\ e_{22} : T_2
\end{aligned}$$

Term rules $\boxed{e_1 \Downarrow_m e_2 : T}$ $\boxed{e_1 \simeq_m e_2 : T}$

$$\begin{aligned}
e_1 \Downarrow_m e_2 : T \iff {} & e_1 \longrightarrow_{\mathsf{C}}^* e_1' \wedge \mathsf{val}_{\mathsf{C}}\ e_1' \wedge \\
& e_2 \longrightarrow_m^* e_2' \wedge \mathsf{val}_m\ e_2' \wedge \\
& e_1' \sim_m e_2' : T
\end{aligned}$$

$$\begin{aligned}
e_1 \simeq_{\mathsf{F}} e_2 : T &\iff e_1 \longrightarrow_{\mathsf{C}}^* \Uparrow l \vee e_1 \Downarrow_{\mathsf{F}} e_2 : T \\
e_1 \simeq_{\mathsf{H}} e_2 : T &\iff (e_1 \longrightarrow_{\mathsf{C}}^* \Uparrow l \wedge e_2 \longrightarrow_{\mathsf{H}}^* \Uparrow l') \vee e_1 \Downarrow_{\mathsf{H}} e_2 : T \\
e_1 \simeq_{\mathsf{E}} e_2 : T &\iff (e_1 \longrightarrow_{\mathsf{C}}^* \Uparrow l \wedge e_2 \longrightarrow_{\mathsf{E}}^* \Uparrow l) \vee e_1 \Downarrow_{\mathsf{E}} e_2 : T
\end{aligned}$$

Type rules $\boxed{T_1 \sim_m T_2}$

$$\begin{aligned}
\{x{:}B \mid e_1\} \sim_m \{x{:}B \mid e_2\} \iff & \\
& \forall e_1' \sim_m e_2' : \{x{:}B \mid \mathsf{true}\}. \\
& \quad e_1[e_1'/x] \simeq_m e_2[e_2'/x] : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \\
T_{11} \to T_{12} \sim_m T_{21} \to T_{22} \iff & T_{11} \sim_m T_{21} \wedge T_{12} \sim_m T_{22}
\end{aligned}$$

Closing substitutions and open terms $\boxed{\Gamma \models_m \delta}$

$\boxed{\Gamma \vdash e_1 \simeq_m e_2 : T}$

$$\begin{aligned}
\Gamma \models_m \delta &\iff \forall x \in \mathsf{dom}(\Gamma).\ \delta_1(x) \sim_m \delta_2(x) : \Gamma(x) \\
\Gamma \vdash e_1 \simeq_m e_2 : T &\iff \forall \Gamma \models_m \delta.\ \delta_1(e_1) \simeq_m \delta_2(e_2) : T
\end{aligned}$$

Figure 11. Modal logical relations relating classic λ_{H} to space-efficient modes

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

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

where

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

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.

As we did for the other calculi, we present the routine syntactic proof of type soundness in the appendix (Section A.5). Like forgetful and heedful λ_{H} before, eidetic λ_{H} shares source programs (Definition 3.1) with classic λ_{H}. With this final lemma, we know that all modes share the same well typed source programs.

7. 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 $\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e_1$ and $\langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e_2$ behave identically. This cast congruence principle is easy to see, because E_CASTINNERC applies freely. In the space-efficient modes, however, E_CASTINNER can only apply when E_CASTMERGE doesn't. Merged casts may not behave the same as running the two casts separately. We summarize the results in commutative diagrams in Figure 10. Forgetful λ_{H} has the property that if the unmerged casts reduce to a value, then so do the merged ones. But the merged casts may reduce to a value when the unmerged ones reduce to blame—because forgetful merging skips checks. Heedful λ_{H} has a stronger property: the merged and unmerged casts *coterminate* at results, if the merged term reduces to blame or a value, so does the unmerged term. If they both go to values, they go to the exact same value; but if they both go to blame, they may blame different labels. This is a direct result of E_CASTMERGE saving only one label from casts. Finally, eidetic λ_{H} has a property as strong as heedful λ_{H}: the merged and unmerged casts coterminate exactly.

It is particularly nice that the key property for relating modes can be proved entirely within each mode, i.e., the cast congruence lemma for forgetful λ_{H} is proved *independently* of classic λ_{H}.

The proofs of cast congruence are in Appendix B, but there are two points worth observing here. First, we need strong normalization to prove cast congruence for heedful λ_{H}: if we reorder checks, we need to know that reordering checks doesn't change the observable behavior. Second, both heedful and eidetic λ_{H} eliminate redundant checks when merging casts, the former by using sets and the latter by means of the $\rhd$ operator. These two calculi show that checking is idempotent: checking a property once is as good as checking it twice—which only holds when checks are pure.

Our proofs relating classic λ_{H} and the space-efficient modes are by (mode-indexed) logical relations, found in Figure 11. The

Forgetful λ_{H}

$$\begin{tikzcd} e_1 \arrow[rr, "\mathsf{F}"'] & & e_2 \end{tikzcd}$$

$$\Downarrow$$

$$\begin{tikzcd} \langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e_1 \arrow[dr, dashed, "\mathsf{F}*"'] & \langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e_2 \arrow[d, "\mathsf{F}*"] \\ & \mathsf{val_F}\ e \end{tikzcd}$$

Heedful λ_{H}

$$\begin{tikzcd} e_1 \arrow[rr, "\mathsf{H}"'] & & e_2 \end{tikzcd}$$

$$\Downarrow$$

$$\begin{tikzcd} \langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e_1 \arrow[d, dashed, "\mathsf{H}*"] & & \langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e_2 \arrow[d, "\mathsf{H}*"] \\ \mathsf{result_H}\ e_1' \arrow[dr, phantom, "="] & \sim & \mathsf{result_H}\ e_2' \arrow[dl, phantom, "="] \\ & \mathsf{val_H}\ e & \end{tikzcd}$$

Eidetic λ_{H}

$$\begin{tikzcd} e_1 \arrow[rr, "\mathsf{E}"'] & & e_2 \end{tikzcd}$$

$$\Downarrow$$

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

Figure 10. Cast congruence lemmas as commutative diagrams

relation is modal: in $e_1 \sim_m e_2 : T$, the term e_1 is a classic λ_{H} term, while e_2 and T are in mode m. Each mode's logical relation matches its cast congruence lemma: the forgetful logical relation allows more blame on the classic side (not unlike the asymmetric logical relations of Greenberg et al. [11]); the heedful logical relation is blame-inexact, allowing classic and heedful λ_{H} to raise different labels; the eidetic logical relation is exact. The proofs can be found in Appendix B. They follow a fairly standard pattern in each mode m: we show that applying C-casts and m-casts between similar and related types to related values yields related values (i.e., casts are applicative); we then show that well typed C-source programs are related to m-source programs. As far as alternative techniques go, an induction over evaluation derivations wouldn't give us enough information about evaluations that return lambda abstractions. But other contextual equivalence techniques (e.g., bisimulation) would probably work, too.

Our equivalence results for forgetful and heedful λ_{H} are subtle: they would break down if we had effects other than blame. Forgetful λ_{H} changes which contracts are checked—and so which code is run; heedful λ_{H} can reorder when code is run. Well typed λ_{H} programs in this paper are strongly normalizing. If we allowed nontermination, for example, then we could construct source programs that diverge in classic λ_{H} and converge in forgetful λ_{H}, or source programs that diverge in one of classic and heedful λ_{H} and converge in the other. Similarly, if blame were a *catchable* exception, we would have no relation for these two modes at all: since they can raise different blame labels, different exception handlers could have entirely different behavior. Eidetic λ_{H} doesn't reorder checks, though, so its result is more durable. As long as checks are pure, eidetic and classic λ_{H} coincide.

One might ask, then, why we bother proving strong results for forgetful and heedful if they only adhere in such a restricted setting? First, we wish to explore the design space—and forgetful and heedful offer insights into the semantics and structure of casts. Second, we want to show soundness of space efficiency in isolation—implementations always differ from the theory. Analagously, languages with first-class stack traces make tail-call optimization observable, but this change in semantics is typically considered worthwhile—space efficiency is more important.

8. Bounds for space efficiency

We have claimed that forgetful, heedful, and eidetic λ_{H} are space efficient: what do we mean? What sort of space efficiency have we achieved in our various calculi? We summarize the results in Table 1; proofs are in Appendix C. Suppose that a type of height h can be represented in W_h bits and a label in L bits. Casts in classic and forgetful λ_{H} each take up $2W_h + L$ bits: two types and a blame label. Casts in heedful λ_{H} take up more space—$2W_h + 2^{W_h} + L$ bits—because they need to keep track of the type set. Coercions

Mode	Cast size	Pending casts
Classic ($m = \mathsf{C}$)	$2W_h + L$	∞
Forgetful ($m = \mathsf{F}$)	$2W_h + L$	$\lvert e \rvert$
Heedful ($m = \mathsf{H}$)	$2W_h + 2^{W_h} + L$	$\lvert e \rvert$
Eidetic ($m = \mathsf{E}$)	$s2^{L+W_1}$	$\lvert e \rvert$

Table 1. Space efficiency of λ_{H}

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. A coercion has a set of refinements and blame labels at each position which take up 2^{L+W_1} space, leading to $s2^{L+W_1}$ space per coercion. A more precise bound might track which refinements appear in which parts of a function type, but in the worst caseit 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. Forgetful, heedful and eidetic λ_{H} can have no more than one pending cast per term node—abstractions are limited to a single function proxy, and E_CASTMERGE 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 sizes 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. Eidetic λ_{H}'s coercions never hold types greater than height 1; the types on its casts are erasable once the coercions are generated—coercions drive the checking.

The bounds we find here are *galactic*—but that is not the point. 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. We have shown that sound space efficiency is possible—it is future work to produce a feasible implementation.

9. Related work

Some earlier work uses first-class casts, whereas our casts are always applied to a term [2, 16]. 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 [24], using coercions [13], casts, casts annotated with intermediate types a/k/a *threesomes*, or some combination of all three [9, 15, 22, 23, 25]. Recent work relates all three frameworks, making particular use of coercions [1]. 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, sometimes allow for simpler proofs, e.g., by induction on evaluation [22]; even when strong reasoning principles are needed, the presence of dynamic types leads them to use bisimulation [1, 9, 25]. 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 [13]. 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 [10], the most closely related work, offers a coercion language combining the dynamic types of Henglein's original work with predicate contracts; his EFFICIENT language is our forgetful λ_{H} without blame. He conjectures that blame for coercions reads left to right (as it does in Siek and Garcia [23]); 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; all of the coercions in our refinement lists are both injections *and* projections. We abstain from using an interrobang '‽' to reduce notation.

Dimoulas et al. [4] 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 more than space efficiency. Findler et al. [7] studied space and time efficiency for datatype contracts.

PLT Racket contracts have a mild form of space efficiency, checking for exact duplicate contracts at tail positions. The redundancy it detects seems to rely on pointer equality. Since PLT Racket contracts are (a) module-oriented "macro" contracts, and (b) first class, this optimization is somewhat unpredictable—and limited compared with our heedful and eidetic calculi.

10. 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. Forgetful λ_{H} is an interesting middle ground: if contracts exist to make partial operations safe (and not abstraction or information hiding), forgetfulness may be a good strategy.

We believe that a latent version of eidetic λ_{H} would not be particularly hard to devise: simply compile contracts into coercions and use our merge operator.

In our simple (i.e., not dependent) case, our refinement types close over a single variable of base type. We can treat these refinement types as interned symbols, for which type comparison is effectively integer comparison. But closure comparisons are notoriously fragile: optimizers may disrupt programmer expectations. Space efficiency for a dependent calculus remains open.

Acknowledgments

Omitted for submission.

References

[1] Anonymous. Blame, coercion, and threesomes: Together again for the first time. In submission., 2014.

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

[3] 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.

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

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

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

[7] 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. doi: 10.1007/978-3-540-85373-2_7.

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

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

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

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

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

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

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

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

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

[17] 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.

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

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

[20] PLT. PLT Racket, 2013. URL `http://racket-lang.org`.

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

[22] 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.

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

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

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

[26] 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.

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

[28] 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.
($T = T_1 \to T_2$) By S_FUN and the IHs.

□

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.

□

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.

□

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. □

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 $\vdash T_1 \parallel T_2$). □

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. □

A.10 Lemma [Reducing type sets]: If $\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_3$ then $\vdash_m (\mathcal{S} \setminus T_2) \parallel T_1 \Rightarrow T_3$.

Proof: All of the $T \in \mathcal{S}$ remain well formed and similar to T_1 and T_3, as do the well formedness and similarity relations for T_1 and T_3. □

A.2 Classic type soundness

A.11 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. □

A.12 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 $T = T_1 \to T_2$, then either $e = \lambda x{:}T.\ e$ or $e = \langle T_{11} \to T_{12} \overset{\bullet}{\Rightarrow} T_{21} \to T_{22} \rangle^l\ e'$.

A.13 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. □

A.14 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. □

A.3 Forgetful type soundness

Just as we did for classic λ_H in Section A.2, we reuse the theorems from Section A.1. Note that if e is a value in forgetful λ_H, it's also a value in classic λ_H, i.e., $\mathsf{val}_{\mathsf{F}}\ e$ implies $\mathsf{val}_{\mathsf{C}}\ e$.

A.15 Lemma [Forgetful determinism]:
If $e \longrightarrow_{\mathsf{F}} e_1$ and $e \longrightarrow_{\mathsf{F}} e_2$ then $e_1 = e_2$.

Proof: By induction on the first evaluation derivation. □

A.16 Lemma [Forgetful canonical forms]: If $\emptyset \vdash_{\mathsf{F}} e : T$ and $\mathsf{val}_{\mathsf{F}}\ e$ then:

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

A.17 Lemma [Forgetful progress]: If $\emptyset \vdash_{\mathsf{F}} e : T$, then either:

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

Proof: By induction on the typing derivation. □

A.18 Lemma [Forgetful preservation]: If $\emptyset \vdash_{\mathsf{F}} e : T$ and $e \longrightarrow_{\mathsf{F}} e'$ then $\emptyset \vdash_{\mathsf{F}} e' : T$.

Proof: By induction on the typing derivation. □

In addition to showing type soundness, we prove that a source program (Definition 3.1) is well typed with $m = \mathsf{F}$ iff it is well typed with $m = \mathsf{C}$.

A.19 Lemma [Source program typing for forgetful λ_H]:
Source programs are well typed in C iff they are well typed in F, i.e.:

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

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

Proof: By mutual induction on e, T, and Γ. □

A.4 Heedful type soundness

A.20 Lemma [Heedful canonical forms]: If $\emptyset \vdash_{\mathsf{H}} e : T$ and $\mathsf{val}_{\mathsf{H}}\ e$ then:

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

A.21 Lemma [Heedful progress]: If $\emptyset \vdash_{\mathsf{H}} e : T$, then either:

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

Proof: By induction on the typing derivation. □

Before proving preservation, we must establish some properties about type sets: type sets as merged by E_CASTMERGE are well formed; the dom and cod operators take type sets of function types and produce well formed type sets.

A.22 Lemma [Merged type sets are well formed]: If $\vdash_{\mathsf{H}} \mathcal{S}_1 \parallel T_1 \Rightarrow T_2$ and $\vdash_{\mathsf{H}} \mathcal{S}_2 \parallel T_2 \Rightarrow T_3$ then $\vdash_{\mathsf{H}} (\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}) \parallel T_1 \Rightarrow T_3$.

Proof: By transitivity of similarity, we have $\vdash T_1 \parallel T_3$. We have $\vdash_{\mathsf{H}} T_1$ and $\vdash_{\mathsf{H}} T_3$ from each of the A_TYPESET derivations, so it remains to show the premises for each $T \in \mathcal{S}$.

Let $T \in (\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\})$. We have $\vdash T \parallel T_1$ and $\vdash_{\mathsf{H}} T$ (a) by assumption and symmetry (Lemma A.5) if $T = T_2$; and (b) by A_TYPESET and symmetry and transitivity (Lemma A.6) if $T \in \mathcal{S}_1 \cup \mathcal{S}_2$. We can therefore apply A_TYPESET, and we are done. □

A.23 Lemma [Domain type set well formedness]: If $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22}$ then $\vdash_{\mathsf{H}} \mathsf{dom}(\mathcal{S}) \parallel T_{21} \Rightarrow T_{11}$.

Proof: First, observe that for every $T \in \mathcal{S}$, we know that $\vdash T \parallel T_{11} \to T_{12}$, so each $T_i = T_{i\,1} \to T_{i\,2}$ by inversion. This means that $\mathsf{dom}(\mathcal{S})$ is well defined.

By inversion of similarity and type well formedness, we have $\vdash T_{11} \parallel T_{21}$ and $\vdash_{\mathsf{H}} T_{11}$ and $\vdash_{\mathsf{H}} T_{21}$. By symmetry of similarity, we have $\vdash T_{21} \parallel T_{11}$ (Lemma A.5).

Let $T_{i\,1} \in \mathsf{dom}(\mathcal{S})$ by given. We know that there exists some $T_{i\,2}$ such that $T_{i\,1} \to T_{i\,2} \in \mathcal{S}$ and $\vdash T_{i\,1} \to T_{i\,2} \parallel T_{11} \to T_{12}$ and $\vdash_{\mathsf{H}} T_{i\,1} \to T_{i\,2}$. By inversion, we find $\vdash T_{i\,1} \parallel T_{11}$ and $\vdash_{\mathsf{H}} T_{i\,1}$. By transitivity of similarity (Lemma A.6), we have $\vdash T_{i\,1} \parallel T_{21}$, and we are done by A_TYPESET. □

A.24 Lemma [Codomain type set well formedness]: If $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22}$ then $\vdash_{\mathsf{H}} \mathsf{cod}(\mathcal{S}) \parallel T_{12} \Rightarrow T_{22}$.

Proof: First, observe that for every $T \in \mathcal{S}$, we know that $\vdash T \parallel T_{11} \to T_{12}$, so each $T_i = T_{i\,1} \to T_{i\,2}$ by inversion. This means that $\mathsf{dom}(\mathcal{S})$ is well defined.

By inversion of similarity and type well formedness, we have $\vdash T_{12} \parallel T_{22}$ and $\vdash_{\mathsf{H}} T_{12}$ and $\vdash_{\mathsf{H}} T_{22}$.

Let $T_{i\,2} \in \mathsf{dom}(\mathcal{S})$ by given. We know that there exists some $T_{i\,2}$ such that $T_{i\,1} \to T_{i\,2} \in \mathcal{S}$ and $\vdash T_{i\,1} \to T_{i\,2} \parallel T_{12} \to T_{12}$ and $\vdash_{\mathsf{H}} T_{i\,1} \to T_{i\,2}$. By inversion, we find $\vdash T_{i\,2} \parallel T_{12}$ and $\vdash_{\mathsf{H}} T_{i\,2}$. We are done by A_TYPESET. □

A.25 Lemma [Heedful preservation]: If $\emptyset \vdash_{\mathsf{H}} e : T$ and $e \longrightarrow_{\mathsf{H}} e'$ then $\emptyset \vdash_{\mathsf{H}} e' : T$.

Proof: By induction on the typing derivation. □

Just as we did for forgetful λ_{H} in (Section A.3), we show that source programs are well typed heedfully iff they are well typed classically—iff they are well typed forgetfull (Lemma A.19). that is, source programs are valid staring points in any mode.

A.26 Lemma [Source program typing for heedful λ_{H}]: Source programs are well typed in C iff they are well typed in H, i.e.:

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

Proof: By mutual induction on e, T, and Γ. □

A.5 Eidetic type soundness

A.27 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. □

A.28 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 $T = T_{21} \to T_{22}$, then either $e = \lambda x{:}T.\ e'$ or $e = \langle T_{11} \to T_{12} \overset{c_1 \mapsto c_2}{\Rightarrow} T_{21} \to T_{22} \rangle^{\bullet}\ \lambda x{:}T_{11}.\ e'$.

A.29 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. □

A.30 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 $\vdash_{\mathsf{E}} \{x{:}B \mid e\}^{l} \rhd r \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}$.

Proof: By cases on the rule used.

(A_REFINE) If r is just the type $\{x{:}B \mid e\}$ with some other blame label, then directly by A_REFINE and the assumption. If not, there is more than one type in r.

If $e = e_2$, then $r \setminus \{x{:}B \mid e\}$ isn't well formed on its own, but adding $\{x{:}B \mid e\}^{l}$ makes it so. If not, then we know that $r \setminus \{x{:}B \mid e\}$ is well formed, and so is its extensions by assumption.

(A_FUN) Contradictory.

□

A.31 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 $\vdash_{\mathsf{E}} c_1 \rhd c_2 \parallel T_1 \Rightarrow T_3$.

Proof: By induction on c_1's typing derivation.

(A_REFINE) By the IH, Lemma A.30, and A_REFINE.
(A_FUN) By the IHs and A_FUN.

□

A.32 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.

□

A.33 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. □

A.34 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. □

B. Proofs of space-efficiency soundness

This appendix contains the proofs relating classic λ_{H} to each other mode: forgetful, heedful, and eidetic.

B.1 Relating classic and forgetful manifest contracts

If we evaluate a λ_{H} term with the classic semantics and find a value, then the forgetful semantics finds a similar value—identical if they're constants. Since forgetful λ_{H} drops some casts, some terms reduce to blame in classic λ_{H} while they reduce to values in forgetful λ_{H}.

The relationship between classic and forgetful λ_{H} is *blame-inexact*, to borrow the terminology of Greenberg et al. [11]: we define an asymmetric logical relation in Figure 12, relating classic values to forgetful values—and everything to classic blame. The proof proceeds largely like that of Greenberg et al. [11]: we define a logical relation on terms and an inductive invariant relation on types, prove that casts between related types are logically related, and then show that well typed source programs are logically related.

Before we explain the logical relation proof itself, there is one new feature of the proof that merits discussion: we need to derive a congruence principle for casts forgetful λ_{H}. When proving that casts between related types are related (Lemma B.2), we want to be able to reason with the logical relation—which involves reducing the cast's argument to a value. But if $e \longrightarrow^*_{\mathsf{F}} e'$ such that $\mathsf{result_F}\ e'$, how to $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e$ and $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e'$ relate? If $e' = \Uparrow l'$ is blame, then it may be that $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e$ reduces to a value while $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l \Uparrow l'$ propagates the blame. But if e' is a value, then both casts reduce to the same value. We show this property first for a single step $e \longrightarrow_{\mathsf{F}} e'$, and then lift it to many steps.

B.1 Lemma [Cast congruence (single step)]: If

– $\emptyset \vdash_{\mathsf{F}} e : T_1$ and and $\vdash_{\mathsf{F}} \emptyset \parallel T_1 \Rightarrow T_2$ (and so $\emptyset \vdash_{\mathsf{F}} \langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e : T_2$),
– $e \longrightarrow_{\mathsf{F}} e_1$ (and so $\emptyset \vdash_{\mathsf{F}} e_1 : T_1$),
– $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e_1 \longrightarrow^*_{\mathsf{F}} e_2$, and
– $\mathsf{val_F}\ e_2$

then $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e \longrightarrow^*_{\mathsf{F}} e_2$.

Proof: By cases on the step $e \longrightarrow_{\mathsf{F}} e_1$. There are three groups of reductions: straightforward merge-free reductions, merging reductions (the interesting cases, where a reduction step taken in e has an exposed cast), and (contradictory) reductions where blame is raised.

None of the blame propagation rules (i.e., E_*RAISE*) can occur: we would have $e_1 = \Uparrow l'$, and then $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l \Uparrow l'$ doesn't reduce to a value, contradicting our assumption. □

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

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

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

$$e_1 \simeq_{\mathsf{F}} e_2 : T \iff e_1 \longrightarrow^*_{\mathsf{C}} \Uparrow l \vee \left(\begin{array}{l} e_1 \longrightarrow^*_{\mathsf{C}} e_1' \wedge \mathsf{val_C}\ e_1' \\ e_2 \longrightarrow^*_{\mathsf{F}} e_2' \wedge \mathsf{val_F}\ e_2' \\ e_1' \sim_{\mathsf{F}} e_2' : T \end{array} \right)$$

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

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

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

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

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

Figure 12. Asymmetric logical relation between classic λ_{H} and forgetful λ_{H}

Once we have cast congruence for a single step, a straightforward induction gives us reasoning principle applicable to many steps.

We define the logical relation in Figure 12. It is defined in a split style, with separate definitions for values and terms. Note that terms that classically reduce to blame are related to all forgetful terms, but terms that classically reduce to values reduce forgetfully to similar values. We lift these closed relations on values and terms to open terms by means of dual closing substitutions. As in Greenberg et al. [11], we define an inductive invariant to relate types, using it to show that casts between related types on related values yield related values, i.e., casts are applicative (Lemma B.2). One important subtle technicality is that the type indices of this logical relation are forgetful types—in the constant case of the value relation, we evaluate the predicate in the forgetful semantics. We believe the choice is arbitrary, but have not tried the proof using classic type indices.

B.2 Lemma [Relating classic and forgetful casts]: If $T_{11} \sim_{\mathsf{F}} T_{21}$ and $T_{12} \sim_{\mathsf{F}} T_{22}$ and $\vdash T_{11} \parallel T_{12}$, then forall $e_1 \sim_{\mathsf{F}} e_2 : T_{21}$, we have $\langle T_{11} \overset{\bullet}{\Rightarrow} T_{12} \rangle^l\ e_1 \simeq_{\mathsf{F}} \langle T_{21} \overset{\bullet}{\Rightarrow} T_{22} \rangle^{l'}\ e_2 : T_{22}$.

Proof: By induction on the sum of the heights of T_{21} and T_{22}. □

B.3 Lemma [Relating classic and forgetful source programs]:

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

Proof: By mutual induction on the typing derivations. □

B.2 Relating classic and heedful manifest contracts

Heedful λ_{H} reorders casts, so we won't necessarily get the same blame as we do in classic λ_{H}. We can show, however, that they blame the same amount: heedful λ_{H} raises blame if and only if classic λ_{H} does, too. We define a blame-inexact, symmetric logical relation.

The proof follows the same scheme as the proof for forgetful λ_H in Section B.1: we first prove a cast congruence principle; then we define a logical relation relating classic and heedful λ_H; we prove a lemma establishing a notion of applicativity for casts using an inductive invariant grounded in the logical relation, and then use that lemma to prove that well typed source programs are logically related.

Cast congruence—that $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e$ and $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e_1$ behave identically when $e \longrightarrow_\mathsf{H} e_1$—holds almost exactly. The pre- and post-step terms may end blaming different labels, but otherwise return identical values. Note that this cast congruence lemma (a) has annotations other than $\bullet$, and (b) is stronger than Lemma B.1, since we not only get the same value out, but we also get blame when the inner reduction yields blame—though the label may be different. The potentially different blame labels in heedful λ_H's cast congruence principle arises because of how casts are merged: heedful λ_H is heedful of types, but forgets blame labels.

B.4 Lemma [First-order casts don't change their arguments]: If $\langle \{x{:}B \mid e_1\} \overset{\mathcal{S}}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^l\ k \longrightarrow^*_\mathsf{H} e$ and $\mathsf{val}_\mathsf{H}\ e$ then $e = k$.

Proof: By induction on the size of $\mathcal{S}$. ☐

B.5 Lemma [Determinism of heedful λ_H]: If $e \longrightarrow_\mathsf{H} e_1$ and $e \longrightarrow_\mathsf{H} e_2$ then $e_1 = e_2$.

Proof: By induction on the first evaluation derivation. In every case, only a single step can be taken. Critically, E_CheckSet uses the choose function, which makes some deterministic choice. ☐

Heedful λ_H's cast congruence proof requires an extra principle. We first show that casting is idempotent: we can safely remove the source type from a type set.

B.6 Lemma [Idempotence of casts]:
If $\emptyset \vdash_\mathsf{H} \langle \{x{:}B \mid e_1\} \overset{\mathcal{S}}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^l\ k : \{x{:}B \mid e_2\}$ and $\emptyset \vdash_\mathsf{H} k : \{x{:}B \mid e_3\}$ then for all $\mathsf{result}_\mathsf{H}\ e$, then:
(a) $\langle \{x{:}B \mid e_1\} \overset{\mathcal{S}}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^l\ k \longrightarrow^*_\mathsf{H} e$ iff
(b) $\langle \{x{:}B \mid e_1\} \overset{\mathcal{S}\setminus\{x:B \mid e_3\}}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^l\ k \longrightarrow^*_\mathsf{H} e$.

Proof: By induction on the size of $\mathcal{S}$, with the terms in lock step until choose produces $\{x{:}B \mid e_3\}$ and we can discharge its check with the fact that $e_3[k/x] \longrightarrow^*_\mathsf{H} \mathsf{true}$. ☐

We need strong normalization to prove cast congruence: if we reorder checks, we need to know that reordering checks doesn't change the observable behavior. We define a unary logical relation to show strong normalization in Figure 13. We assume throughout at the terms are well typed at their indices: $e \in [\![T]\!]$ implies $\emptyset \vdash_\mathsf{H} e : T$ and $\models T$ implies $\vdash_\mathsf{H} T$ and $\models \mathcal{S} \parallel T_1 \Rightarrow T_2$ implies $\vdash_\mathsf{H} \mathcal{S} \parallel T_1 \Rightarrow T_2$ and $\Gamma \models e : T$ implies $\Gamma \vdash_\mathsf{H} e : T$ *by definition*. Making this assumption simplifies many of the technicalities. First, typed terms stay well typed as they evaluate (by preservation, Lemma A.25), so a well typed relation allows us to reason exclusively over typed terms. Second, it allows us to ignore the refinements in our relation, essentially using the simple type structure. After proving cast congruence, we show that all well typed terms are in fact in the relation, i.e., that all heedful terms normalize.

B.7 Lemma [Expansion and contraction]: If $e_1 \longrightarrow^*_\mathsf{H} e_2$ then $e_1 \in [\![T]\!]$ iff $e_2 \in [\![T]\!]$.

Proof: By induction on T.

($T = \{x{:}B \mid e''\}$) By determinism (Lemma B.5).
($T = T_1 \to T_2$) Given some $e' \in [\![T_1]\!]$, we must show that $e_1\ e' \in [\![T_2]\!]$ iff $e_2\ e' \in [\![T_2]\!]$. We have $e_1\ e' \longrightarrow^*_\mathsf{H} e_2\ e'$ by induction on the length of the evaluation derivation and E_AppL, so we are done by the IH on T_2.

☐

Normalizing closed terms $\boxed{e \in [\![T]\!]}$

$$\begin{aligned} e \in [\![\{x{:}B \mid e\}]\!] &\iff e \longrightarrow^*_\mathsf{H} \Uparrow l \lor \\ & e \longrightarrow^*_\mathsf{H} k \land \mathsf{ty}(k) = B \\ e \in [\![T_1 \to T_2]\!] &\iff \forall e' \in [\![T_1]\!].\ \mathsf{result}_\mathsf{H}\ e' \Rightarrow e\ e' \in [\![T_2]\!] \end{aligned}$$

Normalizing open terms $\boxed{\Gamma \models e : T}$ $\boxed{\Gamma \models \sigma}$

$$\begin{aligned} \Gamma \models e : T &\iff \forall \sigma.\ \Gamma \models \sigma \Rightarrow \sigma(e) \in [\![T]\!] \\ \Gamma \models \sigma &\iff \forall x{:}T \in \Gamma.\ \sigma(x) \in [\![T]\!] \end{aligned}$$

Normalizing types and type sets $\boxed{\models T}$ $\boxed{\models \mathcal{S} \parallel T_1 \Rightarrow T_2}$

$$\frac{\forall k.\ \mathsf{ty}(k) = B \text{ implies } e[k/x] \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]}{\models \{x{:}B \mid e\}}\ \text{SWF\_Refine}$$

$$\frac{\models T_1 \quad \models T_2}{\models T_1 \to T_2}\ \text{SWF\_Fun}$$

$$\frac{\vdash T_1 \parallel T_2 \quad \models T_1 \quad \models T_2 \qquad \forall T \in \mathcal{S}.\ \models T \quad \vdash T \parallel T_1}{\models \mathcal{S} \parallel T_1 \Rightarrow T_2}\ \text{SWF\_TypeSet}$$

Figure 13. Strong normalization for heedful λ_H

B.8 Lemma [Blame inhabits every type]: $\Uparrow l \in [\![T]\!]$ for all T.

Proof: By induction on T.

($T = \{x{:}B \mid e''\}$) By definition.
($T = T_1 \to T_2$) By the IH, $\Uparrow l' \in [\![T_1]\!]$. We must show that $\Uparrow l\ \Uparrow l' \in [\![T_2]\!]$. This term steps to $\Uparrow l$ by E_AppRaiseL, and then we are done by contraction (Lemma B.7).

☐

B.9 Lemma [Strong normalization]: If $e \in [\![T]\!]$ then $e \longrightarrow^*_\mathsf{H} e'$ uniquely such that $\mathsf{result}_\mathsf{H}\ e'$.

Proof: Uniqueness is immediate by determinism. We show normalization by induction on T, observing that blame inhabits every type. ☐

B.10 Lemma [Cast congruence (single step)]: If

– $e \in [\![T_1]\!]$ and $\models \mathcal{S} \parallel T_1 \Rightarrow T_2$ (and so $\emptyset \vdash_\mathsf{H} \langle T_1 \overset{\bullet}{\Rightarrow} T_2\rangle^l\ e : T_2$),
– $e \longrightarrow_\mathsf{H} e_1$ (and so $\emptyset \vdash_\mathsf{H} e_1 : T_1$),
– $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e_1 \longrightarrow^*_\mathsf{H} e_2$, and
– $\mathsf{result}_\mathsf{H}\ e_2$

then $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e \longrightarrow^*_\mathsf{H} \Uparrow l'$ if $e_2 = \Uparrow l$ or to e_2 itself if $\mathsf{val}_\mathsf{H}\ e_2$.

Proof: By cases on the step taken; the proof is as for forgetful λ_H (Lemma B.1), though we need to use strong normalization to handle the reorderings. ☐

B.11 Lemma [Cast congruence]: If

– $\emptyset \models e : T_1$ and $\models \mathcal{S} \parallel T_1 \Rightarrow T_2$ (and so $\emptyset \vdash_\mathsf{H} \langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e : T_2$),
– $e \longrightarrow^*_\mathsf{H} e_1$ (and so $\emptyset \vdash_\mathsf{H} e_1 : T_1$),
– $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e_1 \longrightarrow^*_\mathsf{H} e_2$, and
– $\mathsf{result}_\mathsf{H}\ e_2$

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

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

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

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

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

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

Closing substitutions and open terms $\boxed{\Gamma \models_{\mathsf{H}} \delta}$ $\boxed{\Gamma \vdash e_1 \simeq_{\mathsf{H}} e_2 : T}$

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

Figure 14. Blame-inexact, symmetric logical relation between classic λ_{H} and heedful λ_{H}

then $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2 \rangle^l\ e \longrightarrow_{\mathsf{H}}^{*} \Uparrow l'$ if $e_2 = \Uparrow l$ or to e_2 itself if $\mathsf{val}_{\mathsf{H}}\ e_2$.

Proof: By induction on the derivation $e \longrightarrow_{\mathsf{H}}^{*} e_1$, using the single-step cast congruence (Lemma B.10). ☐

B.12 Lemma [Strong normalization of casts]: If $\models \mathcal{S} \parallel T_1 \Rightarrow T_2$ and $e \in [\![T_1]\!]$ then $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2 \rangle^l\ e \in [\![T_2]\!]$.

Proof: By induction on the sum of the heights of T_1 and T_2. ☐

To be able to use our *semantic* cast congruence lemma, we must show that all well typed heedful λ_{H} terms are in the relation we define; this proof is standard.

B.13 Lemma [Strong normalization of heedful terms]:

– $\Gamma \vdash_{\mathsf{H}} e : T$ implies $\Gamma \models e : T$,
– $\vdash_{\mathsf{H}} T$ implies $\models T$, and
– $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_1 \Rightarrow T_2$ implies $\models \mathcal{S} \parallel T_1 \Rightarrow T_2$.

Proof: By mutual induction on the typing derivations. ☐

We define the logical relation in Figure 14; it follows the general scheme of forgetful λ_{H}'s logical relation (Figure 12). The main difference is that this relation is *symmetric*: classic and heedful λ_{H} yield blame or values iff the other one does, thought the blame labels may be different. The formulations are otherwise the same, and the proof proceeds similarly—though heedful λ_{H}'s more complicated cast merging leads to some more intricate stepping in the cast lemma.

B.14 Lemma [Value relation relates only values]: If $e_1 \sim_{\mathsf{H}} e_2 : T$ then $\mathsf{val}_{\mathsf{C}}\ e_1$ and $\mathsf{val}_{\mathsf{H}}\ e_2$.

Proof: By induction on T. We have $e_1 = e_2 = k$ when $T = \{x{:}B \mid e\}$ (and so we are done by V_Const). When $T = T_1 \to T_2$, we have the value derivations as assumptions. ☐

B.15 Lemma [Relation implies similarity]: If $T_1 \sim_{\mathsf{H}} T_2$ then $\vdash T_1 \parallel T_2$.

Proof: By induction on T_1, using S_Refine and S_Fun. ☐

B.16 Lemma [Relating classic and heedful casts]: If $T_{11} \sim_{\mathsf{H}} T_{21}$ and $T_{12} \sim_{\mathsf{H}} T_{22}$ and $\vdash T_{11} \parallel T_{12}$, then forall $e_1 \sim_{\mathsf{H}} e_2 : T_{21}$, we have $\langle T_{11} \overset{\bullet}{\Rightarrow} T_{12} \rangle^l\ e_1 \simeq_{\mathsf{H}} \langle T_{21} \overset{\bullet}{\Rightarrow} T_{22} \rangle^{l'}\ e_2 : T_{22}$.

Proof: By induction on the sum of the heights of T_{21} and T_{22}. ☐

B.17 Lemma [Relating classic and heedful source programs]:

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

Proof: By mutual induction on the typing derivations. ☐

We have investigated two alternatives to the formulation here: type set optimization and invariants that clarify the role of type sets.

First, we can imagine a system that optimizes the type set of $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2 \rangle^l$ such that T_1 and T_2 don't appear in $\mathcal{S}$—taking advantage of idempotence not only for the source type (Lemma B.6) but also for the target type. This change complicates the theory but doesn't give any stronger theorems. Nevertheless, such an optimization would be a sensible addition to an implementation.

Second, our proof relates source programs, which start with empty annotations. In fact, all of the reasoning about type sets is encapsulated in our proof cast congruence. We could define a function from heedful λ_{H} to classic λ_{H} that unrolls type sets according to the choose function. While this proof would offer a direct understanding of heedful λ_{H} type sets in terms of the classic semantics, it wouldn't give us a strong property—it degenerates to our proof in the empty type set case.

B.3 Relating classic and eidetic manifest contracts

B.18 Lemma [Idempotence of coercions]: If $\emptyset \vdash_{\mathsf{E}} k : \{x{:}B \mid e_1\}$ and $\vdash_{\mathsf{E}} r_1 \rhd r_2 \parallel \{x{:}B \mid e_1\} \Rightarrow \{x{:}B \mid e_2\}$, then for all $\mathsf{result}_{\mathsf{E}}\ e$, it is the case that $\langle \{x{:}B \mid e_1\} \overset{r_1 \rhd (r_2 \setminus \{x:B \mid e_1\})}{\Rightarrow} \{x{:}B \mid e_2\} \rangle^{\bullet}\ k \longrightarrow_{\mathsf{E}}^{*} e$ iff $\langle \{x{:}B \mid e_1\} \overset{r_1 \rhd r_2}{\Rightarrow} \{x{:}B \mid e_2\} \rangle^{\bullet}\ k \longrightarrow_{\mathsf{E}}^{*} e$.

Proof: By induction on their evaluation derivations: the only difference is that the latter derivation performs an extra check that $e_1[k/x] \longrightarrow_{\mathsf{E}}^{*} \mathsf{true}$—which we already know to hold. ☐

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

B.19 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 $e_1 \longrightarrow_{\mathsf{E}} e_2$. ☐

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 15. 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.

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

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

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

B.20 Lemma [Casts related to coercions are logically related]: If $T_1 \sim_{\mathsf{E}} T_1'$ and $T_2 \sim_{\mathsf{E}} T_2'$ then for all $e_1 \sim_{\mathsf{E}} e_2 : T_1$, $\langle T_1 \overset{\bullet}{\Rightarrow} T_2 \rangle^l\ e_1 \simeq_{\mathsf{E}} \langle T_1' \overset{\bullet}{\Rightarrow} T_2' \rangle^l\ e_2 : T_2$.

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

B.21 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. □

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 16. 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. □

C.2 Lemma: $\mathsf{types}(T_1) \cup \mathsf{types}(\mathsf{merge}_m(T_1, a_1, T_2, a_2, T_3)) \cup \mathsf{types}(T_3) \subseteq \mathsf{types}(T_1) \cup \mathsf{types}(a_1) \cup \mathsf{types}(T_2) \cup \mathsf{types}(a_2) \cup \mathsf{types}(T_3)$

Proof: By cases on each, but observing that the merged annotation is always no bigger than the original, and that the type T_2 may or may not vanish. □

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

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

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

Type, type set, and coercion type extraction

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

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

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

$$
\begin{array}{rcl}
\mathsf{types}(\bullet) & = & \emptyset \\
\mathsf{types}(\mathcal{S}) & = & \bigcup_{T \in \mathcal{S}} \mathsf{types}(T) \\
\\
\mathsf{types}(\mathsf{nil}) & = & \emptyset \\
\mathsf{types}(\{x{:}B \mid e\}^l, r) & = & \{\{x{:}B \mid e\}\} \cup \mathsf{types}(r) \\
\mathsf{types}(c_1 \mapsto c_2) & = & \mathsf{types}(c_1) \cup \mathsf{types}(c_2)
\end{array}
$$

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

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

Figure 16. Type extraction and type height

Proof: This property is trivial when $a = \bullet$.

When the annotation is a type set, for $\mathsf{dom}(\mathcal{S})$ to be defined, every type in $\mathcal{S}$ must be a function type. So:

$$
\begin{array}{rcl}
\mathsf{types}(\mathcal{S}) & = & \bigcup_{T \in \mathcal{S}} \mathsf{types}(T) \\
& = & \bigcup_{T_1 \to T_2 \in \mathcal{S}} \mathsf{types}(T_1 \to T_2) \\
& = & \bigcup_{T_1 \to T_2 \in \mathcal{S}} \{T_1 \to T_2\} \cup \\
& & \mathsf{types}(T_1) \cup \mathsf{types}(T_2) \\
& \supseteq & \bigcup_{T_1 \to T_2 \in \mathcal{S}} \mathsf{types}(T_1) \\
& = & \bigcup_{T \in \mathsf{types}(\mathsf{dom}(\mathcal{S}))} \mathsf{types}(T) \\
& = & \mathsf{types}(\mathsf{dom}(\mathcal{S}))
\end{array}
$$

Immediate when $a = c_1 \mapsto c_2$. □

C.4 Lemma: $\mathsf{types}(\mathsf{cod}(a)) \subseteq \mathsf{types}(a)$

Proof: Similar to Lemma C.3. □

C.5 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. □

C.6 Lemma [Dropping types doesn't introduce types]: $\mathsf{types}(r \setminus \{x{:}B \mid e\}) \subseteq \mathsf{types}(r)$

Proof: By induction on r.

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

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

□

C.7 Lemma [Coercion merges don't introduce types]: $\mathsf{types}(r_1 \rhd r_2) \subseteq \mathsf{types}(r_1) \cup \mathsf{types}(r_2)$

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.6, we find:

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

□

C.8 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. □