Space-Efficient Manifest Contracts

Michael Greenberg
Princeton University
mg19@cs.princeton.edu

Abstract

The standard algorithm for higher-order contract checking can lead to unbounded space consumption and can destroy tail recursion. In this work, we show how to achieve space efficiency for contract checking. Working in a manifest context, we define a family of languages: classic λ_H, which is inefficient; forgetful λ_H, which is efficient but skips some checks; and heedful λ_H, which is efficient but may change blame labels. We show first that if classic λ_H produces a value, then so does forgetful λ_H (but not vice versa); we then show that classic and heedful λ_H yield identical values, but possibly differing blame labels.

1. Introduction

Types are an extremely successful form of specification: programmers can specify their intent—plus is a function that takes two numbers and returns another number; 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... when the second number isn't zero. Conventional type systems do a good job of stopping many kinds of errors, but partial operations like division and array indexing remain out of reach for most type system's safety guarantees. 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. These techniques are demanding: they can be difficult to understand, they force certain programming idioms, and they often place heavy constraints on the programming language used, requiring purity or even strong normalization.

Contracts are a popular compromise: programmers write type like contracts of the form $\mathsf{Int} \to \{x{:}\mathsf{Int} \mid x \neq 0\} \to \mathsf{Int}$. These type-like specifications can then be checked at runtime. Checking a *predicate contract* 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 contract $T_1 \to T_2$ on a function f is deferred: we wrap f in a *function proxy*. When this function proxy is called with some argument e, we check the domain contract T_1 on e, run the original function f on the result, and then check the codomain contract T_2 on the final result.

Two forms of checking: latent and manifest. We use the manifest here, have some thoughts later on about how to adapt it to latent. The standard algorithm for higher-order contract checking in both the latent and manifest systems can lead to unbounded space consumption and can destroy tail recursion.

Modes

$$
\begin{array}{lcll}
m & ::= & \mathsf{C} & \text{classic } \lambda_\mathrm{H}\text{; Section 3} \\
 & \mid & \mathsf{F} & \text{forgetful } \lambda_\mathrm{H}\text{; Section 4} \\
 & \mid & \mathsf{H} & \text{heedful } \lambda_\mathrm{H}\text{; Section 6}
\end{array}
$$

Types

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

Terms and type sets

$$
\begin{array}{lcl}
e & ::= & x \mid k \mid \lambda x{:}T.\ e \mid e_1\ e_2 \mid op(e_1, \ldots, e_n) \mid \\
 & & \langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2 \rangle^l\ e \mid \mathsf{proxy}^l(T_{11} \to T_{12}, \mathcal{S}, T_{21} \to T_{22}, e) \mid \\
 & & \langle \{x{:}B \mid e_1\}, e_2, k \rangle^l \mid \Uparrow l \\
\mathcal{S} & ::= & \emptyset \mid \{T_1, ..., T_n\}
\end{array}
$$

Figure 1. Syntax of λ_H

Function proxies are the essential problem: they can build up arbitrarily, using unbounded space. And then, when they unfold, they can create an unbounded number of checks on the stack.

To restore space efficiency, we need to restrict the number of function proxies a value can have. This gets us most of the way. But then we also need to make sure that we still can't build up too many checks on the stack—even with bounded function proxies, mutual recursion can lead to unbounded accumulation on the stack.

In this work, we show how to achieve space efficiency for contract checking. To show how the solutions relate, we define a family of languages parameterized by a *mode* m. The core manifest calculus, classic λ_H ($m = \mathsf{C}$) consumes an unbounded amount of space with contracts; *forgetful* λ_H ($m = \mathsf{F}$) consumes a bounded amount of space but sometimes converges to a value when classic λ_H doesn't; and *heedful* λ_H ($m = \mathsf{H}$), which bounds space consumption and behaves almost exactly like classic λ_H (though it may blame different labels).

2. Manifest contracts

λ_H, originally due to Flanagan [2], is the standard manifest contract calculus. We give the syntax for the non-dependent fragment in Figure 1. We have highlighted in yellow the four syntactic forms implementing manifest contracts. This paper paper discusses three modes of λ_H: classic λ_H, mode C; forgetful λ_H, mode F; and heedful λ_H, mode H. Each of these languages will use the syntax of Figure 1, while the typing rules and operational semantics are indexed by the mode m.

The metavariable B is used for base types, of which at least Bool must be present. Types T include refinements of base types $\{x{:}B \mid e\}$ and function types $T_1 \to T_2$. The refinement type $\{x{:}B \mid$

$e\}$ 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.

The terms of λ_H are largely those of the simply-typed lambda calculus: variables, constants k, abstractions, applications, and operations should all be familiar. The first distinguishing feature of λ_H is the *cast*, written $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e$. Here e is term of type T_1, and the cast will perform checks to ensure that e can be treated as a T_2. In this work, we add a *type set* $\mathcal{S}$ to the cast, written over the cast's arrow. In classic and forgetful λ_H, this set will always be empty. In heedful λ_H, we use the type set to combine checks—the key to our space-efficiency strategy.

The three remaining forms—function proxies, active checks, and blame—only occur as the program evaluates. A *function proxy* $\mathsf{proxy}^l(T_{11}\rightarrow T_{12}, \mathcal{S}, T_{21}\rightarrow T_{22}, e)$ implements casts between function types; in all modes, we have E_WrapAbs:

$$\begin{array}{l}\langle T_{11}\rightarrow T_{12} \overset{\mathcal{S}}{\Rightarrow} T_{21}\rightarrow T_{22}\rangle^l\ (\lambda x{:}T_{11}.\ e) \longrightarrow_m \\ \mathsf{proxy}^l(T_{11}\rightarrow T_{12}, \mathcal{S}, T_{21}\rightarrow T_{22}, \lambda x{:}T_{11}.\ e)\end{array}$$

In some formulations of λ_H, function proxies are implemented by introducing a new lambda as a wrapper. This is convenient in applications only ever reduce by β-reduction, but it doesn't suit our purposes at all: in order to prove space efficiency, we need to be able to combine function proxies. While we can imagine a semantics that looks into closures rather than having separate function proxies, we decline to gaze long into the abyss of lambda abstractions, lest they also gaze back into us.

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 will be returned wholesale if the check succeeds. When checks fail, the program raises *blame*, written $\Uparrow l$. Blame is an uncatchable exception thrown when a cast fails.

2.1 Core operational semantics

The operational semantics for our manifest calculi comprises 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. It is more conventional to fix values as a syntactic subset, but that approach would be confusing here: we would need three different metavariables to unambiguously refer to values from each language. The mode-indexed value and result relations neatly avoid any potential confusion between metavariables.

The operational semantics given here is the core, but every calculus needs to add three things: a value rule for function proxies, a congruence rule for casts, and a wrapping rule for function casts applied to function proxies. That is, each calculus will define V_Proxy$m$, E_CastInner$m$, and E_WrapProxy$m$ (among, possibly, other new rules).

The mode-agnostic value rules are straightforward: constants are always values (V_Const), as are lambdas (V_Abs). Each mode defines its own value rule for function proxies, V_Proxy$m$. Results don't depend on the mode: m-values are always m-results (R_Val); blame is always a result, too (R_Blame).

E_Beta applies lambda abstractions via substitution, using a call-by-value rule. Note that β reduction in mode m requires that the argument is an m-value. The reduction rule for operations defers to operations' denotations, $[\![op]\!]$; since these may be partial (e.g., division), we assign types to operations that guarantee totality (see Section 2.2).

E_WrapAbs gives the semantics for casts between function types when applied to abstractions: we wrap them in a function proxy. The type set $\mathcal{S}$ will be empty in classic and forgetful λ_H (i.e., $m \in \{\mathsf{C}, \mathsf{F}\}$). The E_WrapAbs rule is universal, but each mode defines its own rule for function proxies, E_WrapProxy$m$.

E_Unwrap applies function proxies to values. Note that the domain cast is contravariant, while the codomain is covariant. We define operations to extract the domain and codomain type sets from the given type set; since these are only non-empty when $m = \mathsf{H}$, we discuss these operations in greater detail in Section 6.

E_CheckEmpty turns a cast from one refinement type to another into an active check with the same blame label. We discard the source type—we already know that k is a $\{x{:}B \mid e_1\}$—and substitute the scrutinee into the target type, $e_2[k/x]$, as the current state of checking. We must also hold onto the scrutinee, in case the check succeeds. E_CheckInner only applies when the type set is empty. This rule is present in all three systems, and is enough for classic and forgetful λ_H—they only ever have empty type sets. Heedful λ_H adds a rule to handle casts between refinement types with non-empty type sets. Active checks evaluate by the congruence rule E_CheckInner until one of three results adheres: the predicate returns true, so the whole active check returns the scrutinee (E_CheckOK); the predicate returns false, so the whole active check raises blame using the label on the chceck (E_CheckFail); or blame was raised during checking, and we propagate it via E_CheckRaise.

The core semantics includes several other congruence rules: E_AppL, E_AppR, and E_OpInner. Since space bounds rely not only on limiting the number of function proxies but also accumulation of casts on the stack, the core semantics doesn't include a cast congruence rule. Instead, each mode defines its own rule, E_CastInner$m$. In addition to congruence rules, there are blame propagation rules, which are universal: E_AppRaiseL, E_AppRaiseR, E_CastRaise, E_OpRaise.

2.2 Type system

All three modes share a type system, given in Figure 3. Unlike the operational semantics, no calculus adds custom rules—they all use exactly the same rules, parameterized by mode. The type system comprises several relations: context well formedness $\vdash_m \Gamma$ and type well formedness $\vdash_m T$; type compatibility $\vdash T_1 \parallel T_2$, a mode-less comparison of the *skeleton* of two types; type set well formedness $\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2$; and term typing $\Gamma \vdash_m e : T$. The rules thread the mode parameter through the various judgments; the mode is used in the rule for type set well formedness, since only some modes are allowed to have non-empty type sets.

Context well formedness is entirely straightforward; type well formedness requires some care. We establish a priori that the *raw* type $\{x{:}B \mid \mathsf{true}\}$ is well formed for every base type B (WF_Base); we then use raw types to check that refinements are well formed: $\{x{:}B \mid e\}$ is well formed in mode m if e is well typed as a boolean in mode m when x is a value of type B (WF_Refine). Without WF_Base, we'd never get off the ground defining WF_Refine. Function types are well formed in mode m when their domains and codomains are well formed in mode m. (Unlike many recent formulations, ours is not dependent—we leave dependency as future work.)

Type compatibility $\vdash T_1 \parallel T_2$ help determines whether a value of type T_1 can be cast into type T_2. 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 $T_1\rightarrow T_2$. Type compatibility holds whenever T_1 and T_2 share a skeleton: all refinements of the same base type are compatible, and function types are compatible when their domains and codomains are respectively compatible. Type compatibility is reflexive, symmetric, and transitive; i.e., it is an equivalence relation.

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

$$\frac{}{\mathsf{val}_m\ k}\ \textsc{V\_Const} \qquad \frac{}{\mathsf{val}_m\ \lambda x{:}T.\ e}\ \textsc{V\_Abs} \qquad \frac{\mathsf{val}_m\ e}{\mathsf{result}_m\ e}\ \textsc{R\_Val} \qquad \frac{}{\mathsf{result}_m\ \Uparrow l}\ \textsc{R\_Blame}$$

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

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

$$\frac{}{\langle T_{11}\to T_{12} \overset{\mathcal{S}}{\Rightarrow} T_{21}\to T_{22}\rangle^l\ (\lambda x{:}T_{11}.\ e) \longrightarrow_m \mathsf{proxy}^l(T_{11}\to T_{12}, \mathcal{S},\ T_{21}\to T_{22}, \lambda x{:}T_{11}.\ e)}\ \textsc{E\_WrapAbs}$$

$$\frac{\mathsf{val}_m\ e_2}{\mathsf{proxy}^l(T_{11}\to T_{12}, \mathcal{S},\ T_{21}\to T_{22}, e_1)\ e_2 \longrightarrow_{\mathsf{H}} \langle T_{12} \overset{\mathsf{cod}(\mathcal{S})}{\Rightarrow} T_{22}\rangle^l\ (e_1\ (\langle T_{21} \overset{\mathsf{dom}(\mathcal{S})}{\Rightarrow} T_{11}\rangle^l\ e_2))}\ \textsc{E\_Unwrap}$$

$$\mathsf{dom}(\mathcal{S}) = \bigcup_{T\in\mathcal{S}} \mathsf{dom}(T) \qquad \mathsf{cod}(\mathcal{S}) = \bigcup_{T\in\mathcal{S}} \mathsf{cod}(T)$$

$$\frac{}{\langle \{x{:}B \mid e_1\} \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^l\ k \longrightarrow_m \langle \{x{:}B \mid e_2\}, e_2[k/x], k\rangle^l}\ \textsc{E\_CheckEmpty}$$

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

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

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

$$\frac{e_2 \longrightarrow_m e_2'}{\langle \{x{:}B \mid e_1\}, e_2, k\rangle^l \longrightarrow_m \langle \{x{:}B \mid e_1\}, e_2', k\rangle^l}\ \textsc{E\_CheckInner}$$

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

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

Figure 2. Core operational semantics of λ_{H}

We also use type sets, which keep track of pending casts in heedful λ_{H}. The single type set well formedness rule (S_TYPESET) says that $\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2$ irrespective of the m, T_1 and T_2 are well formed in m and compatible. If $m \neq \mathsf{H}$, then $\mathcal{S}$ must be empty; if $m = \mathsf{H}$, then every type in $\mathcal{S}$ is well formed in H and compatible with T_1 (and so T_2, by transitivity). The use of $\Rightarrow$ here is meant to suggest how we use type sets to type casts below. Type set compatibility is stable under removing elements the set $\mathcal{S}$, and it is symmetric and transitive with respective to its type indices (since compatibility itself is symmetric and transitive).

As for term typing, the T_VAR, T_ABS, T_OP, and T_APP rules are entirely conventional. T_BLAME types blame at any (well formed) type. T_CONST allows a constant k to be typed 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 [2–5] and the extra rule of others [1]. We assume that $\mathsf{ty}(k) = \mathsf{Bool}$ implies $k \in \{\mathsf{true}, \mathsf{false}\}$.

We require that $\mathsf{ty}(op)$ only produce well formed first-order types, i.e., types of the form $\vdash_m \{x{:}B_1 \mid e_1\} \to \ \ldots\ \to \{x{:}B_n \mid e_n\}$. We require that the type is consistent with the operations 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 here can't involve casts that stack—because forgetful λ_{H} treats them differently. We believe this is not so stringent a requirement: the types for operations ought to be simple, e.g. $\mathsf{ty}(\mathsf{div}) = \{x{:}\mathsf{Real} \mid \mathsf{true}\}\to\{y{:}\mathsf{Real} \mid y \neq 0\}\to\{z{:}\mathsf{Real} \mid \mathsf{true}\}$, and stacked casts only arise in stack-free terms due to function proxies.

The typing rule for casts defers to the type set well formedness rule: $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^l\ e$ is well formed in mode m when $\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2$ and e is a T_1 (T_CAST).

The final two rules, T_CHECK and T_PROXY, are used for checking active checks and function proxies. These are runtime rules, that should only occur in the midst of evaluation. In fact,

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

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

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

Type compatibility and type set well formedness $\boxed{\vdash T_1 \parallel T_2}$ $\boxed{\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2}$

$$\frac{}{\vdash \{x{:}B \mid e_1\} \parallel \{x{:}B \mid e_2\}}\ \text{P\_REFINE} \qquad \frac{\vdash T_{11} \parallel T_{21} \quad \vdash T_{12} \parallel T_{22}}{\vdash T_{11}{\rightarrow}T_{12} \parallel T_{21}{\rightarrow}T_{22}}\ \text{P\_FUN}$$

$$\frac{\vdash T_1 \parallel T_2 \quad \vdash_m T_1 \quad \vdash_m T_2 \quad \forall T \in \mathcal{S}.\,(m = \mathsf{H} \quad \vdash_{\mathsf{H}} T \quad \vdash T \parallel T_1)}{\vdash_m \mathcal{S} \parallel T_1 \Rightarrow T_2}\ \text{S\_TYPESET}$$

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

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

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

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

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

$$\frac{\vdash_m \Gamma \quad \vdash_m \mathcal{S} \parallel T_{11}{\rightarrow}T_{12} \Rightarrow T_{21}{\rightarrow}T_{22} \quad \emptyset \vdash_m e : T_{11}{\rightarrow}T_{12} \quad \mathsf{val}_m\ \mathsf{proxy}^l(T_{11}{\rightarrow}T_{12}, \mathcal{S}, T_{21}{\rightarrow}T_{22}, e)}{\Gamma \vdash_m \mathsf{proxy}^l(T_{11}{\rightarrow}T_{12}, \mathcal{S}, T_{21}{\rightarrow}T_{22}, e) : T_{21}{\rightarrow}T_{22}}\ \text{T\_PROXY}$$

Figure 3. Typing rules of λ_{H}

they should only ever be applied to closed terms, even though the rules allow for any well formed context. We do this so that we can prove weakening (Lemma 2.2).

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

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

If we are going to prove type soundness via syntactic methods [6], we must have enough information to type k at $\{x{:}B \mid e'\}$. For this reason, T_CHECK requires that $e_1[k/x] \longrightarrow_m^* e_2$; this way, we know that $e'[k/x] \longrightarrow_{\mathsf{C}}^* \mathsf{true}$ at the end of the previous derivation, which is enough to apply T_CONST. The other premises of T_CHECK ensure that the types all match up: that the target refinement type is well formed, that k as the base type in question, and that e_2, the current state of the active check, is also well formed.

Function proxies are casts between function types wrapped around a value. The premises of T_PROXY recapitulate those of T_CAST, though we add a requirement: T_PROXY in mode m can only apply to m-values. This technicality simplifies some of the metatheory later on.

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.

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

- T_CONST assigns the k type $\{x{:}\mathsf{ty}(k) \mid \mathsf{true}\}$.[1] This ensures that any difference between modes of the evaluation relation isn't reflected in the (source program) type system.
- Casts have empty type sets $\mathcal{S} = \emptyset$; i.e., the quantifier in S_TYPESET holds vacuously.
- T_CHECK, T_PROXY, and T_BLAME are not used. Active checks, function proxies, and blame should only arise at runtime.

2.3 Metatheory

One distinct advantage of having a single syntax with parameterized semantics is that some of the metatheory can be done irrespective of the mode. Each mode proves its own canonical forms lemma—since each mode has a unique notion of value—and its own progress and preservation lemmas. But other standard metatheoretical machinery—weakening, substitution, and regularity—can be proved for all modes at once.

2.2 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 $\Gamma_1, x{:}T', \Gamma_2 \vdash_m e : T$.

2.3 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 $\vdash_m \Gamma_1, \Gamma_2$.

[1] We could soundly relax this requirement to allow $\{x{:}\mathsf{ty}(k) \mid e\}$ such that $e[k/x] \longrightarrow_m^* \mathsf{true}$ for any mode m.

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

3. Classic manifest contracts

We give classic λ_H a call-by-value operational semantics in Figure 4. Here we use set the mode to $m = \mathsf{C}$: our evaluation relation will be $\longrightarrow_{\mathsf{C}}$ and we will use typing judgments of the form, e.g. $\Gamma \vdash_{\mathsf{C}} e : T$. We give the value and operational semantics for $m = \mathsf{F}$ in Figure 5. We just give the rules that are relevant for understanding the classic semantics of casts, extending the core rules of Figure 2.

As established in Section 2.1, each mode must define a value rule for proxies, wrapping rule for casts applied to proxies, and a congruence rule for casts. Classic λ_H defines those and nothing else. First, V_PROXYC says that the function proxy

$$\mathsf{proxy}^l(T_{11} \to T_{12}, \emptyset, T_{21} \to T_{22}, e)$$

is a C-value when e is a C-value. That is, function proxies can wrap lambda abstractions and other function proxies alike. The proxy wrapping rule, E_WRAPPROXYC, does just that: when a function proxy reaches a function cast, classic λ_H wraps up another proxy. That is, classic λ_H could really have a single wrapping rule for function types:

$$\frac{\mathsf{val}_{\mathsf{C}}\ e}{\begin{array}{c}\langle T_{11} \to T_{12} \overset{\emptyset}{\Rightarrow} T_{21} \to T_{22} \rangle^l\ e \longrightarrow_{\mathsf{C}} \\ \mathsf{proxy}^l(T_{11} \to T_{12}, \emptyset, T_{21} \to T_{22}, e)\end{array}}$$

Earlier formulations of λ_H typically gave this rule. We split the treatment of abstractions and function proxies here; the distinction is of primary importance when we consider space efficiency.

The congruence rule for casts in classic λ_H, E_CASTINNERH, allows for free use of congruence. In the space efficient calculi, the use of congruence will be limited.

3.1 Type soundness

We can reuse the metatheory from Section 2.3, since the proofs for weakening (Lemma 2.2), regularity (Lemma 2.4), and substitution (Lemma 2.3) are given for *any* mode m. We show type soundness via progress and preservation; to do so, we need a canonical forms lemma for classic λ_H's values.

3.1 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 $e = \mathsf{proxy}^l(T_{11} \to T_{12}, \emptyset, T_{21} \to T_{22}, e)$.

3.2 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. □

3.3 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. □

4. Forgetful space efficiency

In forgetful λ_H, we offer a simple solution to space-inefficient casts: just forget about them. Function proxies will only ever wrap lambda abstractions; trying to wrap a function proxy around a value that already has one will simply throw away the inner proxy. When accumulating casts on the stack, we will 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 type soundness only talks about reduction to values.

Here we use set the mode to $m = \mathsf{F}$: our evaluation relation will be $\longrightarrow_{\mathsf{F}}$ and we will use typing judgments of the form, e.g. $\Gamma \vdash_{\mathsf{F}} e : T$. We give the value and operational semantics for $m = \mathsf{F}$ in Figure 5. We just give the rules that are relevant for understanding the new, forgetful semantics of casts—which profoundly affects how applications evaluate—without recapitulating all of the other rules of Figure 2.

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). The forgetful rule for wrapping fucntion proxies shows that the intermediate type $T_{21} \to T_{22}$ is entirely thrown away (E_WRAPPROXYF). We keep the original inner type, $T_{11} \to T_{12}$, and the new target type, $T_{31} \to T_{32}$. This proxy wrapping rule guarantees that functional values are either lambdas or a single function proxy around a lambda (Lemma 4.1). Limiting the number of function proxies is critical for establishing a space bounds, as we do in Section 8.

The cast congruence rules in forgetful λ_H combine casts in a forgetful way. E_CASTMERGEF takes two casts next to teach other and eliminates the inner type, just like E_WRAPPROXYF did. The congruence rule E_CASTINNERF can only apply when adjoining casts have all been combined: this allows us to put a bound on stack sizes, the other component of establishing space bounds.

Before proving type soundness, we give an overview of why forgetfulness is a sound strategy for space efficiency. The type soundness property typically comprises two properties: (a) well typed programs don't go 'wrong', and (b) well typed terms reduce to values 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—our notion of 'wrong'—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, we λ_H generally relies on subject reduction, property (b). Operators are assigned types that avoid stuckness, i.e., $\mathsf{ty}(op)$ and $[\![op]\!]$ agree. So for, say, integer division, $\mathsf{ty}(\mathsf{div}) = \{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid y \neq 0\} \to \{z{:}\mathsf{Int} \mid \mathsf{true}\}$. 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 2.1) 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_CASTMERGEF:

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

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 see that functions can assume that their inputs meet their input types—a critical property for programmer reasoning.

Just as we did for classic λ_H in Section 3, we reuse the theorems from Section 2.3. Note that the T_PROXY typing rule automatically accommodates our new definition of values. Note further 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 $\mathsf{val}_{\mathsf{C}}\ e$.

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

Values and operational semantics $\boxed{\mathsf{val_C}\ e}$ $\boxed{e_1 \longrightarrow_{\mathsf{C}} e_2}$

$$\frac{\mathsf{val_C}\ e}{\mathsf{val_C}\ \mathsf{proxy}^l(T_{11}\to T_{12}, \emptyset, T_{21}\to T_{22}, e)}\ \text{V\_ProxyC} \qquad \frac{e \longrightarrow_{\mathsf{C}} e'}{\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^l\ e \longrightarrow_{\mathsf{C}} \langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^l\ e'}\ \text{E\_CastInnerC}$$

$$\frac{}{\begin{array}{l}\langle T_{21}\to T_{22} \overset{\emptyset}{\Rightarrow} T_{31}\to T_{32}\rangle^l\ \mathsf{proxy}^{l'}(T_{11}\to T_{12}, \emptyset, T_{21}\to T_{22}, e) \longrightarrow_{\mathsf{C}} \\ \mathsf{proxy}^l(T_{21}\to T_{22}, \emptyset, T_{31}\to T_{32}, \mathsf{proxy}^{l'}(T_{11}\to T_{12}, \emptyset, T_{21}\to T_{22}, e))\end{array}}\ \text{E\_WrapProxyC}$$

Figure 4. Operational semantics of classic λ_{H}

Values and operational semantics $\boxed{\mathsf{val_F}\ e}$ $\boxed{e_1 \longrightarrow_{\mathsf{F}} e_2}$

$$\frac{}{\mathsf{val_F}\ \mathsf{proxy}^l(T_{11}\to T_{12}, \emptyset, T_{21}\to T_{22}, \lambda x{:}T.\ e)}\ \text{V\_ProxyF}$$

$$\frac{}{\begin{array}{l}\langle T_{21}\to T_{22} \overset{\emptyset}{\Rightarrow} T_{31}\to T_{32}\rangle^l\ \mathsf{proxy}^{l'}(T_{11}\to T_{12}, \emptyset, T_{21}\to T_{22}, \lambda x{:}T_{11}.\ e) \longrightarrow_{\mathsf{F}} \\ \mathsf{proxy}^l(T_{11}\to T_{12}, \emptyset, T_{31}\to T_{32}, \lambda x{:}T_{11}.\ e)\end{array}}\ \text{E\_WrapProxyF}$$

$$\frac{e_2 \longrightarrow_{\mathsf{F}} e_2' \quad e_2 \neq \langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l'}\ e_2''}{\langle T_2 \overset{\emptyset}{\Rightarrow} T_3\rangle^l\ e_2 \longrightarrow_{\mathsf{F}} \langle T_2 \overset{\emptyset}{\Rightarrow} T_3\rangle^l\ e_2'}\ \text{E\_CastInnerF} \qquad \frac{}{\langle T_2 \overset{\emptyset}{\Rightarrow} T_3\rangle^l\ (\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l'}\ e_2) \longrightarrow_{\mathsf{F}} \langle T_1 \overset{\emptyset}{\Rightarrow} T_3\rangle^l\ e_2}\ \text{E\_CastMergeF}$$

Figure 5. Operational semantics of forgetful λ_{H}

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

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

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

Proof: By induction on the typing derivation. □

4.3 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 2.1) is well typed with $m = \mathsf{F}$ iff it is well typed with $m = \mathsf{C}$.

4.4 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. □

5. 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 will find 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. [3]: we define an asymmetric logical relation in Figure 6, relating classic values to forgetful values—and everything to classic blame. The proof proceeds largely like that of Greenberg et al. [3]: 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 5.3), 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 $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^l\ e$ and $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^l\ e'$ relate? If $e' = \Uparrow l'$ is blame, then it may be that $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^l\ e$ reduces to a value while $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^l \Uparrow l'$ 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.

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

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

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), and (contradictory) reductions where blame is raised.

Merge-free reductions By E_CASTINNERF and re-application of the rule (one of E_BETA, E_OP, E_UNWRAP, E_APPL, E_APPR, E_CHECKOK, and E_OPINNER).

Merging reductions

(E_CHECKEMPTY) We have $e = (\langle T_3 \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_{11}\}\rangle^{l'}\ k)$ where $T_1 = \{x{:}B \mid e_{11}\}$ and $e_1 = \langle \{x{:}B \mid e_{11}\}, e_{11}[k/x], k\rangle^{l'}$ and $\langle \{x{:}B \mid e_{11}\} \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_1 \longrightarrow^*_{\mathsf{F}} e_2$ such that $\mathsf{val}_{\mathsf{F}}\ e_2$. We must show that $\langle \{x{:}B \mid e_{11}\} \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e \longrightarrow^*_{\mathsf{F}} e_2$.

By inversion of the similarity relation $\vdash \{x{:}B \mid e_{11}\} \parallel T_2$, we know that $T_2 = \{x{:}B \mid e_{12}\}$. If $\langle \{x{:}B \mid e_{11}\} \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_{12}\}\rangle^{l}\ e_1$ reduces to a value, then it must be the case that $e_1 = \langle \{x{:}B \mid e_{11}\}, e_{11}[k/x], k\rangle^{l'} \longrightarrow^*_{\mathsf{F}} k$ and that $e_{12}[k/x] \longrightarrow^*_{\mathsf{F}} \mathsf{true}$ (and so the entire term reduces $\langle \{x{:}B \mid e_{11}\} \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_{12}\}\rangle^{l}\ e_1 \longrightarrow^*_{\mathsf{F}} k = e_2$). If not, we would have gotten $\Uparrow l'$ or $\Uparrow l$.

Instead, we find that:

$$\begin{array}{ll} & \langle \{x{:}B \mid e_{11}\} \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_{12}\}\rangle^{l}\ (\langle T_3 \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_{11}\}\rangle^{l'}\ k) \\ \longrightarrow_{\mathsf{F}} & \langle T_3 \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_{12}\}\rangle^{l}\ k \\ \longrightarrow_{\mathsf{F}} & \langle \{x{:}B \mid e_{12}\}, e_{12}[k/x], k\rangle^{l} \\ \longrightarrow^*_{\mathsf{F}} & \langle \{x{:}B \mid e_{12}\}, \mathsf{true}, k\rangle^{l} \\ \longrightarrow_{\mathsf{F}} & k = e_2 \end{array}$$

(E_WRAPABS) We have:

$$\begin{array}{l} e = \langle T_{31}\to T_{32} \overset{\emptyset}{\Rightarrow} T_{11}\to T_{12}\rangle^{l'}\ (\lambda x{:}T_{31}.\ e') \\ \longrightarrow_{\mathsf{F}} \mathsf{proxy}^{l'}(T_{31}\to T_{32}, \emptyset, T_{11}\to T_{12}, \lambda x{:}T_{31}.\ e') = e_1 \end{array}$$

By the similarity assumptions, we have $T_1 = T_{11}\to T_{12}$ and $T_2 = T_{21}\to T_{22}$ such that $\vdash T_1 \parallel T_2$. We must show that $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_1 \longrightarrow^*_{\mathsf{F}} e_2$ implies $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e \longrightarrow^*_{\mathsf{F}} e_2$.

In a single step, we find that the original term with e_1 reduces:

$$\begin{array}{l} \langle T_{11}\to T_{12} \overset{\emptyset}{\Rightarrow} T_{21}\to T_{22}\rangle^{l} \\ \quad \mathsf{proxy}^{l'}(T_{31}\to T_{32}, \emptyset, T_{11}\to T_{12}, \lambda x{:}T_{31}.\ e') \longrightarrow_{\mathsf{F}} \\ \mathsf{proxy}^{l}(T_{31}\to T_{32}, \emptyset, T_{21}\to T_{22}, \lambda x{:}T_{31}.\ e') = e_2 \end{array}$$

by E_WRAPABS. In the new term with e, we find the reduction:

$$\begin{array}{lr} \langle T_{11}\to T_{12} \overset{\emptyset}{\Rightarrow} T_{21}\to T_{22}\rangle^{l} & \text{(E\_CASTMERGEF)} \\ \quad (\langle T_{31}\to T_{32} \overset{\emptyset}{\Rightarrow} T_{11}\to T_{12}\rangle^{l'}\ (\lambda x{:}T_{31}.\ e')) & \\ \longrightarrow_{\mathsf{F}} \langle T_{31}\to T_{32} \overset{\emptyset}{\Rightarrow} T_{21}\to T_{22}\rangle^{l}\ (\lambda x{:}T_{31}.\ e') & \text{(E\_WRAPABS)} \\ \longrightarrow_{\mathsf{F}} \mathsf{proxy}^{l}(T_{31}\to T_{32}, \emptyset, T_{21}\to T_{22}, \lambda x{:}T_{31}.\ e') = e_2 & \end{array}$$

(E_WRAPPROXYF) We have:

$$\begin{array}{l} e = \langle T_{41}\to T_{42} \overset{\emptyset}{\Rightarrow} T_{11}\to T_{12}\rangle^{l'} \\ \qquad \mathsf{proxy}^{l''}(T_{31}\to T_{32}, \emptyset, T_{41}\to T_{42}, \lambda x{:}T_{31}.\ e') \\ \longrightarrow_{\mathsf{F}} \mathsf{proxy}^{l'}(T_{31}\to T_{32}, \emptyset, T_{41}\to T_{42}, \lambda x{:}T_{31}.\ e') = e_1 \end{array}$$

By the similarity assumptions, T_1 and T_2 are function types $T_{11}\to T_{12}$ and $T_{21}\to T_{22}$ such that $\vdash T_1 \parallel T_2$. We must show that if $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_1$ reduces to $\mathsf{val}_{\mathsf{F}}\ e_2$ then so does $\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e$.

In a single step, we find that the original term with e_1 reduces:

$$\begin{array}{l} \langle T_{11}\to T_{12} \overset{\emptyset}{\Rightarrow} T_{21}\to T_{22}\rangle^{l} \\ \quad \mathsf{proxy}^{l'}(T_{31}\to T_{32}, \emptyset, T_{41}\to T_{42}, \lambda x{:}T_{31}.\ e') \longrightarrow_{\mathsf{F}} \\ \mathsf{proxy}^{l}(T_{31}\to T_{32}, \emptyset, T_{21}\to T_{22}, \lambda x{:}T_{31}.\ e') = e_2 \end{array}$$

by E_WRAPPROXYF. In the new term with e, we find the reduction:

$$\begin{array}{lr} \langle T_{11}\to T_{12} \overset{\emptyset}{\Rightarrow} T_{21}\to T_{22}\rangle^{l} & \text{(E\_CASTMERGEF)} \\ \quad (\langle T_{41}\to T_{42} \overset{\emptyset}{\Rightarrow} T_{11}\to T_{12}\rangle^{l'} & \\ \qquad \mathsf{proxy}^{l''}(T_{31}\to T_{32}, \emptyset, T_{41}\to T_{42}, \lambda x{:}T_{31}.\ e')) & \\ \longrightarrow_{\mathsf{F}} \langle T_{41}\to T_{42} \overset{\emptyset}{\Rightarrow} T_{21}\to T_{22}\rangle^{l} & \text{(E\_WRAPPROXYF)} \\ \qquad \mathsf{proxy}^{l''}(T_{31}\to T_{32}, \emptyset, T_{41}\to T_{42}, \lambda x{:}T_{31}.\ e') & \\ \longrightarrow_{\mathsf{F}} \mathsf{proxy}^{l}(T_{31}\to T_{32}, \emptyset, T_{21}\to T_{22}, \lambda x{:}T_{31}.\ e') = e_2 & \end{array}$$

(E_CASTINNERF) We have:

$$e = \langle T_3 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ e_{11} \longrightarrow_{\mathsf{F}} \langle T_3 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ e_{12} = e_1$$

with $e_{11} \longrightarrow_{\mathsf{F}} e_{12}$ and $e_{11} \neq \langle T_4 \overset{\emptyset}{\Rightarrow} T_3\rangle^{l''}\ e_2''$.

In the original derivation with e_1, we have

$$\begin{array}{rl} \langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ (\langle T_3 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ e_{12}) & \longrightarrow_{\mathsf{F}} \langle T_3 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_{12} \\ & \longrightarrow^*_{\mathsf{F}} e_2 \end{array}$$

by E_CASTMERGEF and then assumption. We find a new derivation with e as follows:

$$\begin{array}{llr} & \langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ (\langle T_3 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ e_{11}) & \text{(E\_CASTMERGEF)} \\ \longrightarrow_{\mathsf{F}} & \langle T_3 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_{11} & \text{(E\_CASTINNER)} \\ & & \text{since } e_{11} \neq \langle T_4 \overset{\emptyset}{\Rightarrow} T_3\rangle^{l''}\ e_2'' \\ \longrightarrow_{\mathsf{F}} & \langle T_3 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_{12} & \text{(assumption)} \\ \longrightarrow^*_{\mathsf{F}} & e_2 & \end{array}$$

(E_CASTMERGEF) We have:

$$e = \langle T_3 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ (\langle T_4 \overset{\emptyset}{\Rightarrow} T_3\rangle^{l''}\ e_{11}) \longrightarrow_{\mathsf{F}} \langle T_4 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ e_{11} = e_1$$

In the original derivation with e_1, we have

$$\langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ (\langle T_4 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ e_{11}) \longrightarrow_{\mathsf{F}} \langle T_4 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_{11} \longrightarrow^*_{\mathsf{F}} e_2$$

We can build a new derivation with e as follows, stepping twice by E_CASTMERGEF:

$$\begin{array}{llr} & \langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ (\langle T_3 \overset{\emptyset}{\Rightarrow} T_1\rangle^{l'}\ (\langle T_4 \overset{\emptyset}{\Rightarrow} T_3\rangle^{l''}\ e_{11})) & \\ \longrightarrow_{\mathsf{F}} & \langle T_3 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ (\langle T_4 \overset{\emptyset}{\Rightarrow} T_3\rangle^{l''}\ e_{11}) & \\ \longrightarrow_{\mathsf{F}} & \langle T_4 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e_{11} & \text{(assumption)} \\ \longrightarrow^*_{\mathsf{F}} & e_2 & \end{array}$$

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

□

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

5.2 Lemma [Cast congruence]: If

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

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

Proof: By induction on the derivation $e \longrightarrow^*_{\mathsf{F}} e_1$, using the single-step cast congruence (Lemma 5.1). □

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

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

Term rules $\boxed{e_1 \gtrapprox e_2 : T}$

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

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

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

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

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

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

Figure 6. Asymmetric logical relation between classic λ_{H} and forgetful λ_{H}

We define the logical relation in Figure 6. 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. [3], 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 5.3). 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.

5.3 Lemma [Relating classic and forgetful casts]: If $T_{11} \gtrsim T_{21}$ and $T_{12} \gtrsim T_{22}$ and $\vdash T_{11} \parallel T_{12}$, then forall $e_1 \gtrsim e_2 : T_{21}$, we have $\langle T_{11} \overset{\emptyset}{\Rightarrow} T_{12} \rangle^l\ e_1 \gtrapprox \langle T_{21} \overset{\emptyset}{\Rightarrow} T_{22} \rangle^{l'}\ e_2 : T_{22}$.

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

5.4 Lemma [Relating classic and forgetful source programs]:

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

Proof: By mutual induction on the typing derivations.

□

6. Heedful space efficiency

Our third and final calculus, 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. It is critical that we use sets: classic λ_{H} allows for arbitrary lists of function proxies and casts on the stack to accumulate. Accumulating a set gives us a straightforward bound on the amount of accumulation: a program of fixed size can only have a certain number of types at each size. We discuss this idea further in Section 8.

Up until this point, the type sets have all been empty. Heedful λ_{H} collects type sets as casts merge to record the types that must be checked. If $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_1 \Rightarrow T_2$, then not only are T_1 and T_2 similar and well formed, but so are all the (well formed) types in $\mathcal{S}$. Broadly, the invariant we're trying to establish is that

$$\langle T_1 \overset{\{T_2\} \cup \mathcal{S}}{\Rightarrow} T_3 \rangle^l\ e \equiv \langle T_2 \overset{\mathcal{S} \setminus T_2}{\Rightarrow} T_3 \rangle^l\ (\langle T_1 \overset{\emptyset}{\Rightarrow} T_2 \rangle^l\ e).$$

We define the operational semantics of heedful λ_{H} in Figure 7. We repeat the E_CheckEmpty rule from the core semantics (Figure 2) so that readers can compare the empty-set case with the non-empty-set case, E_CheckSet. Now that type sets aren't empty, the dom and cod operators on E_Unwrap are nontrivial. These operators and E_Unwrap are defined in Figure 2; they take a type set $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22}$ and produce two type sets: one for the domain $\vdash_{\mathsf{H}} \mathsf{dom}(\mathcal{S}) \parallel T_{21} \Rightarrow T_{11}$ and one for the codomain $\vdash_{\mathsf{H}} \mathsf{cod}(\mathcal{S}) \parallel T_{12} \Rightarrow T_{22}$. Merging casts, as in E_CastMergeH, or a cast and a proxy, as in E_WrapProxyH, requires us to restore the invariants as well: we join a cast (or function proxy) from T_1 to T_2 (with type set $\mathcal{S}_1$) with a cast from T_2 to T_3 (with type set $\mathcal{S}_2$). We form a cast (or function proxy) from T_1 to T_3 with the type set ($\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}$)—we must be careful to include T_2, the intermediate type that would have been dropped by forgetful λ_{H}.

6.1 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 $e = \mathsf{proxy}^l(T_{11} \to T_{12}, \mathcal{S}, T_{21} \to T_{22}, \lambda x{:}T_{11}.\ e')$.

6.2 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_CastMergeH are well formed; the dom and cod operators take type sets of function types and produce well formed type sets. We omit the (straightforward) proofs due to space constraints.

6.3 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 4), we show that source programs are well typed heedfully iff they are well typed classically—iff they are well typed forgetfull (Lemma 4.4). that is, source programs are valid staring points in any mode.

6.4 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. □

As a corollary of this lemma and Lemma 4.4, source programs are well typed in F if and only if they are well typed in H.

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

$$\frac{}{\mathsf{val}_{\mathsf{H}}\ \mathsf{proxy}^{l}(T_{11}\!\to\! T_{12}, \mathcal{S}, T_{21}\!\to\! T_{22}, \lambda x{:}T.\ e)}\ \text{V\_ProxyH}$$

$$\frac{}{\begin{array}{l}\langle T_{21}\!\to\! T_{22} \overset{\mathcal{S}_2}{\Rightarrow} T_{31}\!\to\! T_{32}\rangle^{l_2}\ \mathsf{proxy}^{l_1}(T_{11}\!\to\! T_{12}, \mathcal{S}_1, T_{21}\!\to\! T_{22}, \lambda x{:}T_{11}.\ e) \longrightarrow_{\mathsf{H}} \\ \mathsf{proxy}^{l_2}(T_{11}\!\to\! T_{12}, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_{21}\!\to\! T_{22}\}, T_{31}\!\to\! T_{32}, \lambda x{:}T_{11}.\ e)\end{array}}\ \text{E\_WrapProxyH}$$

$$\frac{}{\langle \{x{:}B \mid e_1\} \overset{\emptyset}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^{l}\ k \longrightarrow_{m} \langle \{x{:}B \mid e_2\}, e_2[k/x], k\rangle^{l}}\ \text{E\_CheckEmpty}$$

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

$$\frac{e_2 \longrightarrow_{\mathsf{H}} e_2' \quad e_2 \neq \langle T_1 \overset{\mathcal{S}'}{\Rightarrow} T_2\rangle^{l'}\ e_2''}{\langle T_2 \overset{\mathcal{S}}{\Rightarrow} T_3\rangle^{l}\ e_2 \longrightarrow_{\mathsf{H}} \langle T_2 \overset{\mathcal{S}}{\Rightarrow} T_3\rangle^{l}\ e_2'}\ \text{E\_CastInnerH} \qquad \frac{}{\langle T_2 \overset{\mathcal{S}_2}{\Rightarrow} T_3\rangle^{l_2}\ (\langle T_1 \overset{\mathcal{S}_1}{\Rightarrow} T_2\rangle^{l_1}\ e) \longrightarrow_{\mathsf{H}} \langle T_1 \overset{\mathcal{S}_1 \cup \mathcal{S}_2 \cup \{T_2\}}{\Rightarrow} T_3\rangle^{l_2}\ e}\ \text{E\_CastMergeH}$$

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

Figure 7. Operational semantics of heedful λ_{H}

7. 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 will define a blame-inexact, symmetric logical relation.

The proof follows the same scheme as the proof for forgetful λ_{H} in Section 5: 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 will otherwise return identical values. Note that this cast congruence lemma (a) has potentially non-empty type sets, and (b) is stronger than Lemma 5.1, since we not only get the same value out, but we will 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. We can imagine an "eidetic" λ_{H} that associates labels with types, though we don't think we can obtain a stronger theorem: eidetic λ_{H} would still have to reorder checks, which could result blaming a label earlier than the classic semantics would. We leave the investigation of eidetic λ_{H} as future work.

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.

7.1 Lemma [Idempotence of casts]: If

$$\emptyset \vdash_{\mathsf{H}} \langle \{x{:}B \mid e_1\} \overset{\mathcal{S}}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^{l}\ k : \{x{:}B \mid e_2\}$$

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) $\langle \{x{:}B \mid e_1\} \overset{\mathcal{S}\setminus\{x:B|e_1\}}{\Rightarrow} \{x{:}B \mid e_2\}\rangle^{l}\ k \longrightarrow_{\mathsf{H}}^{*} e$.

Proof: By induction on the size of $\mathcal{S}$, with the terms in lock step until choose produces $\{x{:}B \mid e_1\}$ and we can discharge its check with the fact that $e_1[k/x] \longrightarrow_{\mathsf{H}}^{*} \mathsf{true}$. □

7.2 Lemma [Cast congruence (single step)]: If

- $\emptyset \vdash_{\mathsf{H}} e : T_1$ and $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_1 \Rightarrow T_2$ (and so $\emptyset \vdash_{\mathsf{H}} \langle T_1 \overset{\emptyset}{\Rightarrow} T_2\rangle^{l}\ e : T_2$),
- $e \longrightarrow_{\mathsf{H}} e_1$ (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. There are two groups of reductions: straightforward merge-free reductions and merging reductions.

Merge-free reductions In these cases, we apply E_CastInnerH and whatever rule derived $e \longrightarrow_{\mathsf{H}} e_1$. (Every rule not involving merges: E_Beta, E_Op, E_Unwrap, E_AppL, E_AppR, E_AppRaiseL, E_AppRaiseR, E_CheckOK, E_CheckFail, E_OpInner, and E_OpRaise)

Merging reductions In these cases, some cast in e reduces when we step $e \longrightarrow_{\mathsf{H}} e_1$, but merges when we consider $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^{l}\ e$. We must show that the merged term and $\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2\rangle^{l}\ e_1$ eventually meet. After merging the cast in e—and possibly some steps in e_1— the e and e_1 terms will reduce to a common term, which immediately gives us the common reduction to results we need. (E_CheckEmpty, E_CheckSet, E_WrapAbs, E_WrapProxy, E_CastInnerH, E_CastMergeH, and E_CastRaise) □

7.3 Lemma [Cast congruence]: If

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

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

Term rules $\boxed{e_1 \simeq e_2 : T}$

$$e_1 \simeq e_2 : T$$
$$\iff$$
$$\begin{pmatrix} e_1 \longrightarrow^*_{\mathsf{C}} \Uparrow l \wedge \\ e_2 \longrightarrow^*_{\mathsf{H}} \Uparrow l' \end{pmatrix} \vee \begin{pmatrix} e_1 \longrightarrow^*_{\mathsf{C}} e_1' \wedge \mathsf{val}_{\mathsf{C}}\ e_1' \wedge \\ e_2 \longrightarrow^*_{\mathsf{H}} e_2' \wedge \mathsf{val}_{\mathsf{H}}\ e_2' \wedge \\ e_1' \sim e_2' : T \end{pmatrix}$$

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

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

Closing substitutions and open terms $\boxed{\Gamma \models \delta}$ $\boxed{\Gamma \vdash e_1 \simeq e_2 : T}$

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

Figure 8. Blame-inexact, symmetric logical relation between classic λ_{H} and heedful λ_{H}

- $\emptyset \vdash_{\mathsf{H}} e : T_1$ and $\vdash_{\mathsf{H}} \mathcal{S} \parallel T_1 \Rightarrow T_2$ (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
- $\mathsf{val}_{\mathsf{H}}\ e_2$

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

Proof: By induction on the derivation $e \longrightarrow^*_{\mathsf{H}} e_1$, using the single-step cast congruence (Lemma 7.2). □

We define the logical relation in Figure 8; it follows the general scheme of forgetful λ_{H}'s logical relation (Figure 6). 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.

7.4 Lemma [Relating classic and heedful casts]: If $T_{11} \gtrsim T_{21}$ and $T_{12} \gtrsim T_{22}$ and $\vdash T_{11} \parallel T_{12}$, then forall $e_1 \sim e_2 : T_{21}$, we have $\langle T_{11} \overset{\emptyset}{\Rightarrow} T_{12} \rangle^l\ e_1 \simeq \langle T_{21} \overset{\emptyset}{\Rightarrow} T_{22} \rangle^{l'}\ e_2 : T_{22}$.

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

7.5 Lemma [Classic and heedful source programs are related]:

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

Proof: By mutual induction on the typing derivations. □

If we design 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 7.1) but also for the target type—then we get a more complicated theory, but no stronger theorems.

Our result here is subtle—its tightness depends on the fact that we only have one effect, blame. Heedful λ_{H} reorders effects: if we had nontermination, for example, then we could find divergence on one side and blame on the other. When our only effect is blame, we can have a strong theorem, as above. When we have other effects, like nontermination, the reordered effects may be less easy to characterize than "identical up to blame labels". Similarly, if blame were a *catchable* exception, we would have no relation between classic λ_{H} and the space-efficient forgetful and heedful λ_{H} at all: since they can raise different blame labels, different exception handlers could have entirely different behavior. We give this stronger proof (in a more restrictive system) not because we advocate pure, strongly normalizing languages without catchable exceptions, but because it shows that space efficiency in isolation is a sound implementation of manifest contracts. Once this soundness has been established, whether an inefficient classic implementation would differ for a given program is less relevant when the reference implementation has a heedful semantics. By way of a final comparison, languages with first-class stack traces offer an observable way to differentiate tail recursion, but this is typically considered worthwhile—space efficiency is more important.

8. Bounds for space efficiency

We have claimed that forgetful and heedful 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. 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. Classic λ_{H} can have an infinite number of "pending casts"—casts and function proxies—in a program. Forgetful and heedful λ_{H} can have no more than one pending cast per term node—abstractions are restricted to having a single function proxy, and E_CASTMERGEF and E_CASTMERGEH merge adjacent pending casts.

In order to prove space efficiency, we define a function collecting all of the distinct types that appear in a program (Figure 9). 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 $\{y{:}\mathsf{Int} \mid y \neq 0\}$. The text of a program e is finite, so the set $\mathsf{types}(e)$ 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).

8.1 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. □

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. This is, in general, an over-approximation: the source, target, and type set types must all be compatible, which means they are of the same height:

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

Compatible types have the same height, so we can 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—all types in the program are of height 1—we revert to the original bound. In all of these examples, types are

Mode	Cast size	Pending casts	Reduction behavior
Classic ($m = \mathsf{C}$)	$2W_h + L$	∞	Normative
Forgetful ($m = \mathsf{F}$)	$2W_h + L$	$\lvert e \rvert$	$\longrightarrow^*_{\mathsf{C}}$ $\mathsf{val} \Rightarrow \longrightarrow^*_{\mathsf{F}}$ val (Lemma 5.4)
Heedful ($m = \mathsf{H}$)	$2W_h + 2^{W_h} + L$	$\lvert e \rvert$	$\longrightarrow^*_{\mathsf{C}}$ $\mathsf{result} \Leftrightarrow \longrightarrow^*_{\mathsf{H}}$ result (Lemma 7.5)

Table 1. Summary of λ_{H} behavior by mode

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

$$
\begin{array}{rcl}
\mathsf{types}(x) & = & \emptyset \\
\mathsf{types}(k) & = & \emptyset \\
\mathsf{types}(\lambda x{:}\,T.\ e) & = & \mathsf{types}(T) \cup \mathsf{types}(e) \\
\mathsf{types}(\langle T_1 \overset{\mathcal{S}}{\Rightarrow} T_2 \rangle^l\ e) & = & \mathsf{types}(T_1) \cup \mathsf{types}(T_2) \cup \\
 & & \mathsf{types}(\mathcal{S}) \cup \mathsf{types}(e) \\
\mathsf{types}(e_1\ e_2) & = & \mathsf{types}(e_1) \cup \mathsf{types}(e_2) \\
\mathsf{types}(op(e_1, \ldots, e_n)) & = & \bigcup_{1 \le i \le n} \mathsf{types}(e_i) \\
\mathsf{types}(\mathsf{proxy}^l(T_{11} \to T_{12}, \mathcal{S}, T_{21} \to T_{22}, e)) & = & \mathsf{types}(T_{11} \to T_{12}) \cup \\
 & & \mathsf{types}(T_{21} \to T_{22}) \cup \mathsf{types}(\mathcal{S}) \\
\mathsf{types}(\langle \{x{:}B \mid e_1\}, e_2, k \rangle^l) & = & \mathsf{types}(\{x{:}B \mid e_1\}) \cup \mathsf{types}(e_2) \\
\mathsf{types}(\Uparrow l) & = & \emptyset
\end{array}
$$

Type and type set type extraction

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

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

Figure 9. Type extraction

effectively interned symbols, and type comparison is effectively integer comparison.

Acknowledgments

Comments from Rajeev Alur, Ron Garcia, Fritz Henglein, Greg Morrisett, Stephanie Weirich, Phil Wadler, and Steve Zdancewic improved a previous version of this work based on coercions. Some comments from Benjamin Pierce led me to realize that a cast formulation was straightforward.

This work was supported in part by the NSF under grant 0915671 *Contracts for Precise Types* and by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are the author's and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

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

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

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

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

[5] 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.

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