Contracts Made Manifest

Michael Greenberg, Benjamin C. Pierce, Stephanie Weirich
University of Pennsylvania

August 31, 2009

Abstract

Since Findler and Felleisen [2002] introduced *higher-order contracts*, many variants of their system have been proposed. Broadly, these fall into two groups: some follow Findler and Felleisen in using *latent* contracts, purely dynamic checks that are transparent to the type system; others use *manifest* contracts, where *refinement types* record the most recent check that has been applied. These two approaches are generally assumed to be equivalent—different ways of implementing the same idea, one retaining a simple type system, and the other providing more static information. Our goal is to formalize and clarify this folklore understanding.

Our work extends that of Gronski and Flanagan [2007], who defined a latent calculus λ_{C} and a manifest calculus λ_{H}, gave a translation ϕ from λ_{C} to λ_{H}, and proved that if a λ_{C} term reduces to a constant, then so does its ϕ-image. We enrich their account with a translation ψ in the opposite direction and prove an analogous theorem for ψ.

More importantly, we generalize the whole framework to *dependent contracts*, where the predicates in contracts can mention variables from the local context. This extension is both pragmatically crucial, supporting a much more interesting range of contracts, and theoretically challenging. We define dependent versions of λ_{C} (following Findler and Felleisen's semantics) and λ_{H}, establish type soundness—a challenging result in itself, for λ_{H}—and extend ϕ and ψ accordingly. Interestingly, the intuition that the two systems are equivalent appears to break down here: we show that ψ preserves behavior exactly, but that a natural extension of ϕ to the dependent case will sometimes yield terms that blame more because of a subtle difference in the treatment of dependent function contracts when the codomain contract itself abuses the argument.

1 Introduction

The idea of contracts—arbitrary program predicates acting as dynamic pre- and post-conditions—was popularized by Eiffel [Meyer, 1992]. More recently, Findler and Felleisen [2002] introduced a λ-calculus with *higher-order contracts*. This calculus includes terms like $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}\rangle^{l,l'}\,1$, in which a boolean predicate (pos) is applied to a run-time value (1). This term evaluates to 1 since the predicate returns true in this case. On the other hand, the term $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}\rangle^{l,l'}\,0$ evaluates to the *blame term* $\Uparrow l$, signaling that a contract with label l has been violated. The other label on the contract, l', comes into play with *function contracts*, $c_1 \mapsto c_2$. For example, the term

$$\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\} \mapsto \{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}\rangle^{l,l'}\,(\lambda x{:}\mathsf{Int}.\ \mathsf{pred}\,x)$$

"wraps" the function $\lambda x{:}\mathsf{Int}.\ \mathsf{pred}\,x$ in a pair of checks: whenever the wrapped function is called, the argument is checked to see whether it is nonzero and, if not, the blame term $\Uparrow l'$ is produced, signaling that the *context* of the contracted term violated the expectations of the contract. If the argument check succeeds, then the function is run and its result is checked against the contract $\mathsf{pos}\,x$, raising $\Uparrow l$ if this fails (e.g., if the wrapped function is applied to 1).

Findler and Felleisen's work sparked a resurgence of interest in contracts, and in the intervening years a bewildering variety of related systems have been studied. Broadly, these systems come in two different sorts. In systems with *latent* contracts, types and contracts do not interact. Examples of this style include Findler and Felleisen's original system, Hinze et al. [2006], Blume and McAllester [2006], Chitil and Huch [2007], Guha et al. [2007], and Tobin-Hochstadt and Felleisen [2008]. On the other hand, *manifest* contracts play a significant role in the type system, which tracks, for each value, the most recently checked contract. *Hybrid types* [Flanagan, 2006] are a well-known example in this style; others include the work of Ou et al. [2004], Wadler and Findler [2009], and Gronski et al. [2006].

The key feature of manifest systems is that expressions like $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\}$ are incorporated into the type system as *refinement types*. Values of refinement type are introduced via *casts* like $\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\}\rangle^{l}\,n$, which has static type $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\}$ and checks, dynamically, that n is actually nonzero, raising $\Uparrow l$ if it is not. Similarly, $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}\rangle^{l}\,n$ casts an integer that is statically known to be nonzero to one that is statically known to be positive.

Casts between functions types are the analogue of function contracts in the manifest world. For example, consider

$$f \quad = \quad \langle I \to I \Rightarrow P \to P\rangle^{l}\,(\lambda x{:}I.\ \mathsf{pred}\,x)$$

where $I = \{x{:}\mathsf{Int} \mid \mathsf{true}\}$ and $P = \{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}$. The sequence of events when f is applied to some argument n (of type P) is similar to what we saw before: first, n is cast from P to I (it happens that this cast cannot fail, since the target predicate is just true, but if it did it would raise $\Uparrow l$); then the function body is evaluated, and finally the result is cast from I to P, raising $\Uparrow l$ if the cast fails.

One interesting point here is that the blame label l is the same in both cases. This difference is not essential—both latent and manifest contract systems can be defined using more or less rich algebras of blame—but rather a question of the pragmatics of assigning responsibility for failures. Informally, a function contract check $\langle c_1 \mapsto c_2\rangle^{l,l'}\,f$ is dividing up responsibility for f's behavior between its body and its environment: the programmer is saying "If f is ever applied to an argument that doesn't pass c_1, I refuse responsibility ($\Uparrow l'$), whereas if f's result for good arguments doesn't satisfy c_2, I accept responsibility ($\Uparrow l$)." On the other hand, in a manifest system, the programmer who writes the cast $\langle R_1 \to R_2 \Rightarrow S_1 \to S_2\rangle^{l}\,f$ is saying "Although all I know about f is that its results satisfy R_2 when it is applied to arguments satisfying R_1, I assert that it's OK to use it on arguments satisfying S_1 (because I believe that S_1 implies R_1) and I assert that its results will then always satisfy S_2 (because R_2 implies S_2)." In the latter case, the programmer is taking responsibility for *both* assertions (so $\Uparrow l$ makes sense in both cases), while the additional responsibility for checking that arguments satisfy S_1 will be discharged elsewhere (with another cast, with a different blame label).

While contract checks in latent systems intuitively seem to be doing much the same thing as typecasts involving refinement types in manifest systems, the formal correspondence is not obvious. This has led to some confusion in the community about the essential mechanisms of contracts. And, as we will see, matters become yet murkier as we consider richer languages with features such as dependency.

Gronski and Flanagan [2007] initiated a formal investigation of the connection between the latent and manifest worlds. They defined a core calculus, λ_{C}, capturing the essence of latent contracts in the setting of the simply typed lambda-calculus, and an analogous manifest core calculus λ_{H}. To compare these systems, they introduced ϕ, a type-preserving translation from λ_{C} to λ_{H}. What makes ϕ interesting is that it homomorphically maps the analogous features of the two systems: contracts over base types are mapped to casts at base type, and function contracts are mapped to function casts. Their main result is that ϕ preserves behavior, in the sense that if a term t in λ_{C} evaluates to a final result k, then so does its translation $\phi(t)$.

Our work extends theirs in two directions. First, we strengthen their main result by introducing a new homomorphic translation, ψ, from λ_{H} back to λ_{C}, and proving a similar behavioral correspondence theorem for ψ. (We also give a new, more detailed, proof of their correspondence theorem for ϕ.) This shows that the manifest and latent approaches are effectively equivalent in the nondependent case.

Second, and more significantly, we extend the whole story to allow dependency in function contracts in λ_{C} and in arrow types in λ_{H}. Dependency is particularly well-suited to contracts, as it allows for very precise specifications of how the results of functions depend on their arguments. For example, here is a contract that we might want to use as a run-time sanity check for an implementation of vector concatenation:

$$z_1{:}\mathsf{Vec} \mapsto z_2{:}\mathsf{Vec} \mapsto \{z_3{:}\mathsf{Vec} \mid \mathsf{vlen}\, z_3 \ = \ \mathsf{vlen}\, z_1 \ + \ \mathsf{vlen}\, z_2\}$$

Adding dependent contracts to λ_{C} is not too hard: the dependency is all in the contracts and the types stay simple. In λ_{H}, though, dependency significantly complicates the metatheory, requiring the addition of a denotational semantics for types and kinds to break a potential circularity in the definitions, plus an intricate sequence of technical lemmas involving parallel reduction to establish type soundness. (Although Gronski and Flanagan worked only with nondependent λ_{C} and λ_{H}, Knowles and Flanagan [2009] later showed soundness for a variant of dependent λ_{H} in which order of evaluation is non-deterministic and failed casts get stuck instead of raising blame. We discuss the relation between their development and ours in Section 7.)

Moreover, in the dependent case, the tight correspondence between λ_{C} and λ_{H} breaks down a little, in the sense that a natural generalization of the translations does not preserve blame exactly. We can show an exact correspondence for ψ, but there are λ_{C} terms that terminate at values while their ϕ-images in λ_{H} go to blame.[1] The reason for this discrepancy is contracts like

$$f{:}(N \mapsto I) \mapsto \{z{:}\mathsf{Int} \mid f\,0 \ = \ 0\}$$

where $I = \{x{:}\mathsf{Int} \mid \mathsf{true}\}$ and $N = \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}$. This rather strange contract has the basic shape $f{:}c_1 \mapsto c_2$, where c_2 uses f in a way that violates c_1! In particular, if we apply it to $\lambda f{:}\mathsf{Int} \to \mathsf{Int}.\ 0$ and then apply the result to $\lambda x{:}\mathsf{Int}.\ x$ and 5, the final result will be 5, since $\lambda x{:}\mathsf{Int}.\ x$ does satisfy the contract $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}$ and 5 satisfies the contract $\{z{:}\mathsf{Int} \mid (\lambda x{:}\mathsf{Int}.\ x)\, 0 \ = \ 0\}$. However, the translation of f into λ_{H} inserts an extra check, wrapping the occurrence of f in the codomain contract with a cast from $N \to I$ to $I \to I$, which fails when the wrapped function is applied to 0. We discuss this in greater detail in Section 6.

In summary, our main contributions are (a) the translation ψ and a symmetric version of Gronski and Flanagan's behavioral correspondence theorem, (b) the basic metatheory of (CBV, blame-sensitive) dependent λ_{H}, (c) dependent versions of ϕ and ψ and their properties, and (d) a weaker version of the behavioral correspondence in the dependent case.

2 The nondependent languages

As a warm-up, we begin with the nondependent versions of λ_{C} and λ_{H} and (in the next section) the translations between them. The dependent languages, dependent translations, and their properties are

[1] There could, in principle, be some other way of defining ϕ that (a) preserves types, (b) is maps base contracts to refinement-type casts and function contracts to arrow-type casts, and (c) induces an exact behavioral equivalence even in the dependent case, but we have experimented unsuccessfully with a number of alternatives. We conjecture that no such ϕ exists.

$$\begin{array}{rcl} B & ::= & \mathsf{Bool} \mid \ldots \\ k & ::= & \mathsf{true} \mid \mathsf{false} \mid \ldots \end{array}$$

Figure 1: Base types and constants for λ_{C} and λ_{H}

Types and contracts

$$\begin{array}{rcl} T & ::= & B \mid T_1 \to T_2 \\ c & ::= & \{x{:}B \mid t\} \mid c_1 \mapsto c_2 \end{array}$$

Terms, values, results, and evaluation contexts

$$\begin{array}{rcl} t & ::= & x \mid k \mid \lambda x{:}T_1.\ t_2 \mid t_1\, t_2 \mid \\ & & \Uparrow l \mid \langle c \rangle^{l,l'} \mid \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} \\ v & ::= & k \mid \lambda x{:}T_1.\ t_2 \mid \langle c \rangle^{l,l'} \mid \langle c_1 \mapsto c_2 \rangle^{l,l'}\ v \\ r & ::= & v \mid \Uparrow l \\ E & ::= & [\,]\ t \mid v\ [\,] \mid \langle \{x{:}B \mid t\}, [\,], k \rangle^{l} \end{array}$$

Figure 2: Syntax for λ_{C}

$$\frac{}{k\ v \longrightarrow_c [\![k]\!](v)} \quad \text{E\_Const}$$

$$\frac{}{(\lambda x{:}T_1.\ t_2)\ v \longrightarrow_c t_2\{x := v\}} \quad \text{E\_Beta}$$

$$\frac{}{\langle \{x{:}B \mid t\} \rangle^{l,l'}\ k \longrightarrow_c \langle \{x{:}B \mid t\}, t\{x := k\}, k \rangle^{l}} \quad \text{E\_CCheck}$$

$$\frac{}{\langle \{x{:}B \mid t\}, \mathsf{true}, k \rangle^{l} \longrightarrow_c k} \quad \text{E\_OK}$$

$$\frac{}{\langle \{x{:}B \mid t\}, \mathsf{false}, k \rangle^{l} \longrightarrow_c \Uparrow l} \quad \text{E\_Fail}$$

$$\frac{}{(\langle c_1 \mapsto c_2 \rangle^{l,l'}\ v)\ v' \longrightarrow_c \langle c_2 \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v'))} \quad \text{E\_CDecomp}$$

$$\frac{t_1 \longrightarrow_c t_2}{E\,[t_1] \longrightarrow_c E\,[t_2]} \quad \text{E\_Compat}$$

$$\frac{}{E\,[\Uparrow l] \longrightarrow_c \Uparrow l} \quad \text{E\_Blame}$$

Figure 3: Operational semantics for λ_{C}

developed in Sections 4, 5, and 6.

The language λ_C

The language λ_C is the simply-typed lambda calculus straightforwardly augmented with contracts. The most interesting feature is the *contract* term $\langle c \rangle^{l,l'}$ that, when applied to a term t, dynamically ensures that t and its surrounding context satisfy c.[2] If t doesn't satisfy c, then the *positive* label l will be blamed and the whole term will reduce to $\Uparrow l$; on the other hand, if the context doesn't treat $\langle c \rangle^{l,l'}$ t as c demands, then the negative label l' will be blamed and the term will reduce to $\Uparrow l'$. There are two forms of contracts: base contracts $\{x{:}B \mid t\}$ over a base type B and higher-order contracts $c_1 \mapsto c_2$, which check the arguments and results of functions.

The syntax of λ_C appears in Figures 1 and 2. Besides the contract term $\langle c \rangle^{l,l'}$, it includes first-order constants k, blame, and *active checks* $\langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l}$. Active checks do not appear in source programs; they are present only to support the small-step operational semantics, as we explain below. Also, note that we only allow contracts over base types B. We have function contracts like $\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \mapsto \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}$, but not contracts over functions like $\{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\, \mathsf{true} = f\, \mathsf{false}\}$. We discuss this point further in Section 8.

Values v comprise abstractions, contracts, function contracts applied to values, and constants; a *result* r is either a value or $\Uparrow l$ for some l. We define constants using three constructions: the set $\mathcal{K}_B$, which containts constants of base type B; the type-assignment function $\mathrm{ty_c}$, which maps constants to first-order types of the form $B_1 \to B_2 \to \ldots \to B_n$ (and which is assumed to agree with $\mathcal{K}_B$); and the denotation function $[\![-]\!]$ which maps constants to functions from constants to constants (or blame, to allow for partiality). Denotations must agree with $\mathrm{ty_c}$, i.e., if $\mathrm{ty_c}(k) = B_1 \to B_2$, then $[\![k]\!](k_1) \in \mathcal{K}_{B_2}$ if $k_1 \in \mathcal{K}_{B_1}$). We assume that Bool is among the base types, with $\mathcal{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

The operational semantics is given in Figure 3. It includes six rules for basic (small-step, call-by-value) reductions, plus two rules that involve evaluation contexts E (Figure 2). The evaluation contexts implement a left-to-right evaluation order for function application. If $\Uparrow l$ ever appears in the active position of an evaluation context, it is propagated to the top level. As usual, values (and results) do not step.

The first two basic rules are standard, implementing primitive reductions and β-reductions for abstractions. In these rules, arguments must be values v. Since constants are first-order, we know that $v = k'$ in E_Const for well-typed applications.

The next four rules, E_CCheck, E_OK, E_Fail and E_CFun, describe the semantics of contracts. In E_CCheck, base-type contracts applied to constants step to an active check. Active checks include the original contract, the current state of the check, the constant being checked, and a label to blame if necessary. If the check evaluates to true, then E_OK returns the initial constant. If false, the check has failed and a contract has been violated, so E_Fail steps the term to $\Uparrow l$. Higher-order contracts on a value v wait to be applied to an additional argument. When that argument has also been reduced to a value v', E_CDecomp decomposes the function cast: the argument value is checked with the argument part of the contract (switching positive and negative blame, since the context is responsible for the argument), and the result of the application is checked with the result contract.

The typing rules for λ_C (Figure 4) are straightforward, assigning expressions simple types. We give types to constants using the type-assignment function $\mathrm{ty_c}$. Blame expressions have all types. Contracts are checked for well-formedness using the judgment $\vdash_c c : T$, comprising the rules T_BaseC, which requires that the checking term in a base contract return a boolean value when supplied with a term of the right type, and T_FunC. Note that the predicate t in a contract $\{x{:}B \mid t\}$ can contain at most x free, since we are talking about non-dependent contracts here. Contract application, like function application, is checked using T_App.

The T_Checking rule only applies in the empty context—all that is needed because active checks are a technical device that should not be used in source programs. The rule ensures that the contract

[2] Our presentation differs slightly from that of Gronski and Flanagan [2007], since we use first-class contracts $\langle c \rangle^{l,l'}$ rather than forcing all contracts to be applied, as in $t^{c,l,l'}$; details of how ϕ changes can be found in Section 6.

$\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \textsc{T\_Var}$$

$$\frac{}{\Gamma \vdash k : \mathrm{ty_c}(k)} \quad \textsc{T\_Const}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2} \quad \textsc{T\_Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \quad \textsc{T\_App}$$

$$\frac{\vdash_c c : T}{\Gamma \vdash \langle c \rangle^{l,l'} : T \to T} \quad \textsc{T\_Contract}$$

$$\frac{}{\Gamma \vdash \Uparrow l : T} \quad \textsc{T\_Blame}$$

$$\frac{\begin{array}{cc} \emptyset \vdash k : B & \emptyset \vdash t_2 : \mathsf{Bool} \\ \vdash_c \{x{:}B \mid t_1\} : B & \vdash t_2 \supset t_1\{x := k\} \end{array}}{\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B} \quad \textsc{T\_Checking}$$

$\boxed{\vdash_c c : T}$

$$\frac{x{:}B \vdash t : \mathsf{Bool}}{\vdash_c \{x{:}B \mid t\} : B} \quad \textsc{T\_BaseC}$$

$$\frac{\vdash_c c_1 : T_1 \qquad \vdash_c c_2 : T_2}{\vdash_c c_1 \mapsto c_2 : T_1 \to T_2} \quad \textsc{T\_FunC}$$

$\boxed{\vdash t_2 \supset t_1}$

$$\frac{t_1 \longrightarrow_c^* \mathsf{true} \quad \text{implies} \quad t_2 \longrightarrow_c^* \mathsf{true}}{\vdash t_1 \supset t_2} \quad \textsc{T\_Imp}$$

Figure 4: Typing rules for λ_{C}

Types

$$\begin{array}{lll} S & ::= & \{x{:}B \mid s_1\} \ \mid\ S_1 \to S_2 \end{array}$$

Terms, values, results, and evaluation contexts

$$\begin{array}{lll} s & ::= & x \ \mid\ k \ \mid\ \lambda x{:}S_1.\ s_2 \ \mid\ s_1\, s_2 \ \mid \\ & & \Uparrow l \ \mid\ \langle S_1 \Rightarrow S_2\rangle^l \ \mid\ \langle \{x{:}B \mid s_1\}, s_2, k\rangle^l \\ w & ::= & k \ \mid\ \lambda x{:}S_1.\ s_2 \ \mid\ \langle S_1 \Rightarrow S_2\rangle^l \ \mid \\ & & \langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22}\rangle^l\ w \\ q & ::= & w \ \mid\ \Uparrow l \\ F & ::= & [\,]\ s \ \mid\ w\ [\,] \ \mid\ \langle \{x{:}B \mid s\}, [\,], k\rangle^l \end{array}$$

Figure 5: Syntax for λ_{H}

$\{x{:}B \mid t_1\}$ has the right base type for the constant k, that the check expression t_2 has a boolean type, and that the check is actually checking the right contract. The latter condition is formalized by the T_Imp rule: $\vdash\ t_2 \supset t_1\{x := k\}$ asserts that if t_2 evaluates to true, then the original check $t_1\{x := k\}$ must also evaluate to true. This requirement is needed for two reasons: first, nonsensical terms like $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}, \mathsf{true}, 0\rangle^l$ should not be well typed; and second, we use this property in showing that the translations are type preserving (see Theorem 5.11 for ψ and Section 6 for ϕ). This rule obviously makes typechecking for the full "internal language" with checks undecidable, but excluding checks recovers decidability.

The language enjoys standard preservation and progress theorems. Together, these ensure that evaluating a well-typed term to a normal form always yields a result r, which is either blame or a value.

The language λ_{H}

Our second core calculus, nondependent λ_{H}, notably includes *refinement types* and *cast expressions.* The syntax appears in Figure 5. Unlike λ_{C}, which separates contracts from types, λ_{H} combines them into refined base types $\{x{:}B \mid s_1\}$ and normal function types $S_1 \to S_2$. As in Section 2, we do not allow refinement types over functions, nor do we allow refinements of refinements (which add no expressive power if conjunction is available). Unrefined base types B are *not* valid types; they must be written with a trivial refinement, as the *raw* type $\{x{:}B \mid \mathsf{true}\}$. The terms of the language are mostly standard, including variables, the same first-order constants as λ_{C}, blame, abstractions, and applications. The cast expression $\langle S_1 \Rightarrow S_2\rangle^l$ dynamically checks that a given term of type S_1 can be given type S_2. Like λ_{C}, this language uses active checks to give a small-step semantics to cast expressions. Note that we only use a single label for casts, following the original formulation of hybrid types Flanagan [2006].

The values of λ_{H} comprise constants, abstractions, casts, and function casts applied to values. Results q are either values or blame. We give meaning to constants as we did in λ_{C}, reusing the denotation function $[\![-]\!]$. Type assignment is via ty_{h} (which we assume produces well-formed types, defined in Figure 7). To keep the languages in sync, we require that ty_{h} and ty_{c} agree on "type skeletons": if $\mathrm{ty}_{\mathrm{c}}(k) = B_1 \to B_2$, then $\mathrm{ty}_{\mathrm{h}}(k) = \{x{:}B_1 \mid s_1\} \to \{x{:}B_2 \mid s_2\}$.(We will place some further requirements on s_1 and s_2 when we relate the two in detail, in Section 3.)

The small-step, call-by-value semantics in Figure 6 comprises six basic rules and two rules involving evaluation contexts F. Each rule corresponds closely to its counterpart in λ_{C}.

It is worth observing how the decomposition rules compare. In λ_{C}, the term $(\langle c_1 \mapsto c_2\rangle^{l,l'}\ v)\ v'$ decomposes in a straightforward way: c_1 checks the argument v' and c_2 checks the result of the application. In λ_{H} the term $(\langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22}\rangle^l\ w)\ w'$ decomposes to two casts. The contravariant check $\langle S_{21} \Rightarrow S_{11}\rangle^l\ w'$ makes w' a suitable input for w, while $\langle S_{12} \Rightarrow S_{22}\rangle^l$ checks the result from w applied to (the cast) w'. Suppose $S_{21} = \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$ and $S_{11} = \{x{:}B \mid \mathsf{nonzero}\, x\}$. Then the check on the argument ensures that $\mathsf{nonzero}\, x\ \longrightarrow_h^* \mathsf{true}$—not, as one might expect, that $\mathsf{pos}\, w'\ \longrightarrow_h^* \mathsf{true}$. While it is easy to read off from a λ_{C} contract exactly which checks will occur at runtime, a λ_{H} cast must be dissected carefully to

$$\frac{}{k\ w \longrightarrow_h [\![k]\!](w)} \quad \textsc{F\_Const}$$

$$\frac{}{(\lambda x{:}S_1.\ s_2)\ w_2 \longrightarrow_h s_2\{x := w_2\}} \quad \textsc{F\_Beta}$$

$$\frac{}{\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\ k \longrightarrow_h \langle\{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l} \quad \textsc{F\_CCheck}$$

$$\frac{}{\langle\{x{:}B \mid s\}, \mathsf{true}, k\rangle^l \longrightarrow_h k} \quad \textsc{F\_OK}$$

$$\frac{}{\langle\{x{:}B \mid s\}, \mathsf{false}, k\rangle^l \longrightarrow_h \Uparrow l} \quad \textsc{F\_Fail}$$

$$\frac{}{\begin{array}{l}(\langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22}\rangle^l\ w)\ w' \longrightarrow_h \\ \quad \langle S_{12} \Rightarrow S_{22}\rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w'))\end{array}} \quad \textsc{F\_CDecomp}$$

$$\frac{s_1 \longrightarrow_h s_2}{F\,[s_1] \longrightarrow_h F\,[s_2]} \quad \textsc{F\_Compat}$$

$$\frac{}{F\,[\Uparrow l] \longrightarrow_h \Uparrow l} \quad \textsc{F\_Blame}$$

Figure 6: Operational semantics for λ_{H}

see exactly which checks will take place. On the other hand, which label will be blamed when a contract fails is more obvious with casts. The translations below will need to reconcile these facts, carefully encoding correct checking and blame behavior.

The typing rules for λ_{H} (Figure 7) are also similar to those of λ_{C}. Just as the λ_{C} rule T_Contract checks to make sure that the contract has the right form, the λ_{H} rule S_Cast ensures that the two types in a cast expression have the same simple-type skeletons.

$$\begin{array}{rcl}\lfloor \{x{:}B \mid s\} \rfloor & = & B \\ \lfloor S_1 \to S_2 \rfloor & = & \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor\end{array}$$

The S_Cast rule also requires that the types in the cast are well formed, using the type well-formedness judgment $\vdash S$.

Type well-formedness is similar to contract well-formedness in λ_{C}, though the WF_Raw case is added to get us off the ground.

The active check rule S_Checking plays a role analogous to the T_Checking rule in λ_{C}, using the operational S_Imp rule to guarantee that we only have sensible terms in the predicate position.

An important difference between λ_{C} and λ_{H} is that λ_{H} has subtyping. The S_Sub rule allows an expression to be promoted to any well-formed supertype. Refinement types are supertypes if, for all constants of the base type, their condition evaluates to true whenever the subtype's condition evaluates to true. For function types, we use the standard subtyping rule: covariant on the right and contravariant on the left.

We do not consider source programs with subtyping, since it makes the type system undecidable. Rather,

$\boxed{\Delta \vdash s : S}$

$$\frac{x{:}S \in \Delta}{\Delta \vdash x : S} \qquad \textsc{S\_Var}$$

$$\frac{}{\Delta \vdash k : \mathrm{ty_h}(k)} \qquad \textsc{S\_Const}$$

$$\frac{\vdash S_1 \qquad \Delta, x{:}S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x{:}S_1.\ s_2 : S_1 \to S_2} \qquad \textsc{S\_Lam}$$

$$\frac{\Delta \vdash s_1 : S_1 \to S_2 \qquad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\, s_2 : S_2} \qquad \textsc{S\_App}$$

$$\frac{\vdash S_1 \qquad \vdash S_2 \qquad \lfloor S_1 \rfloor = \lfloor S_2 \rfloor}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2} \qquad \textsc{S\_Cast}$$

$$\frac{\vdash S}{\Delta \vdash \Uparrow l : S} \qquad \textsc{S\_Blame}$$

$$\frac{\Delta \vdash s : S_1 \qquad \vdash S_2 \qquad \vdash S_1 <: S_2}{\Delta \vdash s : S_2} \qquad \textsc{S\_Sub}$$

$$\frac{\begin{array}{c}\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\} \qquad \emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \\ \vdash \{x{:}B \mid s_1\} \qquad \vdash s_2 \supset s_1\{x := k\}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}} \qquad \textsc{S\_Checking}$$

$\boxed{\vdash S_1 <: S_2}$

$$\frac{\forall k \in \mathcal{K}_B.\ \vdash s_1\{x := k\} \supset s_2\{x := k\}}{\vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}} \qquad \textsc{Sub\_Base}$$

$$\frac{\vdash S_{21} <: S_{11} \qquad \vdash S_{12} <: S_{22}}{\vdash S_{11} \to S_{12} <: S_{21} \to S_{22}} \qquad \textsc{Sub\_Fun}$$

$\boxed{\vdash s_1 \supset s_2}$

$$\frac{s_1 \longrightarrow_h^* \mathsf{true} \quad \text{implies} \quad s_2 \longrightarrow_h^* \mathsf{true}}{\vdash s_1 \supset s_2} \qquad \textsc{S\_Imp}$$

$\boxed{\vdash S}$

$$\frac{}{\vdash \{x{:}B \mid \mathsf{true}\}} \qquad \textsc{WF\_Raw}$$

$$\frac{x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\vdash \{x{:}B \mid s\}} \qquad \textsc{WF\_Refine}$$

$$\frac{\vdash S_1 \qquad \vdash S_2}{\vdash S_1 \to S_2} \qquad \textsc{WF\_Fun}$$

Figure 7: Typing rules for λ_{H}

it is a technical device for ensuring that types are preserved during reduction.[3] Consider the term:

$$\langle \{x{:}\mathsf{Int} \mid x = 1\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \rangle^{l}\ 1 \ \longrightarrow_{h}^{*}\ 1$$

We would like this term to be well typed, so we must have $\Delta \vdash 1 : \{x{:}\mathsf{Int} \mid x = 1\}$. Note that the cast term has type $\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$. Since it evaluates to 1, we also need $\Delta \vdash 1 : \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$. Whatever we set $\mathrm{ty_h}(1)$ to, it must be a subtype of $\{x{:}\mathsf{Int} \mid s\}$ *whenever* $s\{x := 1\} \longrightarrow_{h}^{*} \mathsf{true}$. That is, constants of base type must have "most-specific" types. One instantiation of this requirement for any $k \in \mathcal{K}_B$ is to set $\mathrm{ty_h}(k) = \{x{:}B \mid x = k\}$; then if $s\{x := k\} \longrightarrow_{h}^{*} \mathsf{true}$, we have $\vdash \mathrm{ty_h}(k) <: \{x{:}B \mid s\}$.

Standard progress and preservation theorems also hold for λ_{H}. We can see them as a restriction of the progress and preservation theorems for dependent λ_{H} (Theorem 4.33 and Theorem 4.32, respectively).

3 The nondependent translations

The latent and manifest calculi differ in a few respects. Obviously, λ_{C} uses contract application and λ_{H} uses casts. Second, λ_{C} contracts have two labels—one positive, one negative—where λ_{H} contracts have a single label. Finally, λ_{H} has a much richer type system than λ_{C}. Both translations—our ψ from λ_{H} to λ_{C} and Gronski and Flanagan's ϕ from λ_{C} to λ_{H}—must account for these differences. We sketch the main ideas of both of these translations.

The interesting parts of the translations deal with contracts and casts. Everything else is translated homomorphically: variables, constants, and blame are left alone; applications and active checks are translated compatibly; lambdas are translated compatibly, though we must choose the type annotation carefully.

For ψ, translating from λ_{H}'s rich types to λ_{C}'s simple types is easy: we just erase the types to their simple skeletons. The interesting case is how we translate the cast $\langle S_1 \Rightarrow S_2 \rangle^{l}$ to the contract $\langle \psi(S_1, S_2) \rangle^{l,l}$ by translating the pair of types together. We define ψ as two mutuall recursive functions: $\psi(s)$ translates λ_{H} terms to λ_{C} terms; $\psi(S_1, S_2)$ translates a pair of λ_{H} types—effectively, a cast—to a λ_{C} contract. The latter function is defined as follows:

$$\begin{array}{rcl} \psi(\{x{:}B \mid s_1\}, \{x{:}B \mid s_2\}) & = & \{x{:}B \mid \psi(s_2)\} \\ \psi(S_{11} \to S_{12}, S_{21} \to S_{22}) & = & \psi(S_{21}, S_{11}) \mapsto \psi(S_{12}, S_{22}) \end{array}$$

We use the single label on the cast in both the positive and negative positions of the resulting contract. When we translate a pair of refinement types, we produce a contract that will check the predicate of the target type (like F_CCheck); when translating a pair of function types, we translate the domain contravariantly (like F_CDecomp). For example, consider the cast:

$$\begin{array}{c} \langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \\ \{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\, y\} \rangle^{l} \end{array}$$

It translates to the contract $\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{pos}\, y\} \rangle^{l,l}$.

In the reverse direction, from λ_{C} to λ_{H}, we are translating from a simple type system to a very rich one. The translation ϕ (which is essentially the same as Gronski and Flanagan's) generates terms in λ_{H} with *raw* types—λ_{H} types with trivial refinements. These are essentially simple types, so they correspond well with the simple types of λ_{C}. We define an operator $\lceil - \rceil$ that maps types and contracts to λ_{H} types:

$$\begin{array}{rcl} \lceil \{x{:}B \mid t\} \rceil & = & \{x{:}B \mid \mathsf{true}\} \\ \lceil c_1 \mapsto c_2 \rceil & = & \lceil c_1 \rceil \to \lceil c_2 \rceil \end{array}$$

Since the translation targets raw types, the type preservation theorem is stated as "if $\Gamma \vdash t : T$ then $\lceil \Gamma \rceil \vdash \phi(t) : \lceil T \rceil$" (see Section 6).

[3] Trade-offs between static subtype checking and dynamic predicate checking that allow decidable systems with subtyping are discussed in detail in Flanagan [2006] and Knowles and Flanagan [2009].

Whereas the difficulty with ψ is ensuring that the checks match up, the difficulty with ϕ is ensuring that the terms in λ_{C} and λ_{H} will blame the same labels. We deal with this problem by translating a single contract with two blame labels into two separate casts. Intuitively, the cast carrying the negative blame label will run all of the checks in negative positions in the contract, while the cast with the positive blame label will run the positive checks. We let

$$\phi(\langle c\rangle^{l,l'}) = \lambda x{:}\lceil c\rceil.\ \langle \phi(c) \Rightarrow \lceil c\rceil\rangle^{l'}\ (\langle \lceil c\rceil \Rightarrow \phi(c)\rangle^{l}\ x),$$

where the translation of contracts to refined types is:

$$\begin{array}{rcl} \phi(\{x{:}B \mid t\}) & = & \{x{:}B \mid \phi(t)\} \\ \phi(c_1 \mapsto c_2) & = & \phi(c_1) \to \phi(c_2) \end{array}$$

The operation of casting into and out of raw types can be thought of as a kind of "bulletproofing." Bulletproofing maintains the raw-type invariant: the positive cast takes x out of $\lceil c\rceil$ and the negative cast puts it back in. For example, consider this contract:

$$\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{pos}\, y\}\rangle^{l,l'}$$

It translates to this λ_{H} term:

$$\begin{array}{l} \lambda f{:}\lceil \mathsf{Int} \to \mathsf{Int}\rceil. \\ \quad \langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\, y\} \Rightarrow \lceil \mathsf{Int} \to \mathsf{Int}\rceil\rangle^{l'} \\ \quad\ (\langle \lceil \mathsf{Int} \to \mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\, y\}\rangle^{l}\ f) \end{array}$$

The domain of the negative check will check that f's argument is nonzero with $\langle \lceil \mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}\rangle^{l'}$. The domain of the positive check will do nothing, since the cast $\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \Rightarrow \lceil \mathsf{Int}\rceil\rangle^{l}$ has no effect. Similarly, the codomain of the negative cast does nothing while the codomain of the positive cast will check that the result is positive. Separating the checks allows λ_{H} to keep track of blame labels, mimicking λ_{C}. This embodies the idea of contracts as pairs of projections Findler [2006]. Note that bulletproofing is *not* necessary at base type. For example, we translate $\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}\rangle^{l,l'}$ to this:

$$\begin{array}{l} \lambda x{:}\lceil \mathsf{Int}\rceil. \\ \quad \langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \Rightarrow \lceil \mathsf{Int}\rceil\rangle^{l'} \\ \quad\ (\langle \lceil \mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}\rangle^{l}\ x) \end{array}$$

Only the positive case does anything—casting into raw types always succeeds. This asymmetry makes sense when you realize that the negative label l' has no purpose on contracts of base type, either.

These translations preserve behavior in a strong sense: if $\Gamma \vdash t : B$, then either t and $\phi(t)$ both evaluate to the same constant k or they both raise $\Uparrow l$ for the same l; and conversely for ψ. Interestingly, we need to set up this behavioral correspondence *before* we can prove that the translations preserve well-typedness, because of the T_Checking and S_Checking rules.

4 The dependent languages

We now extend λ_{C} to dependent function contracts and λ_{H} to dependent functions. The changes are summarized in Figures 8 and 13. Very little needs to be changed in λ_{C}, since contracts and types barely interact; the changes to E_CDecomp and T_FunC are the important ones. Adding dependency to λ_{H} is more involved. In particular, adding contexts to the subtyping judgment entails adding contexts to S_Imp. To avoid a dangerous circularity, we must define closing substitutions in terms of a separate type semantics. Additionally, the new F_CDecomp rule has a somewhat unintuitive (but necessary) asymmetry, as we explain in Section 4.

Contracts and contexts

$$\begin{array}{rcl} c & ::= & \{x{:}B \mid t\} \;\mid\; x{:}c_1 \mapsto c_2 \\ \Gamma & ::= & \emptyset \;\mid\; \Gamma, x{:}T \;\mid\; \Gamma, x{:}c \end{array}$$

Operational Semantics

$$\frac{}{(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\; v)\; v' \;\longrightarrow_c\; \langle c_2\{x := v'\}\rangle^{l,l'}\; (v\; (\langle c_1\rangle^{l',l}\; v'))} \quad \text{E\_CDecomp}$$

Typing rules

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x \;:\; T} \quad \text{T\_VarT}$$

$$\frac{x{:}c \in \Gamma}{\Gamma \vdash x \;:\; \lfloor c \rfloor} \quad \text{T\_VarC}$$

$$\frac{\Gamma \vdash_c c_1 \;:\; T_1 \qquad \Gamma, x{:}c_1 \vdash_c c_2 \;:\; T_2}{\Gamma \vdash_c x{:}c_1 \mapsto c_2 \;:\; T_1 \to T_2} \quad \text{T\_FunC}$$

Figure 8: Changes for dependent λ_C

Dependent λ_C

Dependent λ_C has been studied since Findler and Felleisen [2002]; it received a very thorough treatment (in the untyped case) in Blume and McAllester [2006], was ported to Haskell by Hinze et al. [2006] and Chitil and Huch [2007], and is used as a specification language in Xu et al. [2009]. Type soundness is not particularly difficult, since types and contracts are kept separate. Our formulation follows Findler and Felleisen [2002], with a few technical changes to make the proofs for ϕ easier.

The only changes to the system of Section 2 to add dependency to λ_C are the new T_FunC and E_CDecomp rules. We also make some small changes to the bindings in contexts: T_FunC adds $x{:}c_1$ to the context when checking the codomain of a function contract, and we split the variable rule in two. Both of these changes help ϕ preserve types. (See Section 6).

Two different E_CDecomp rules can be found in the literature: we call them the *lax* and *picky* variants. The original rule in Findler and Felleisen [2002] is *lax* (like ours, and like most other contract calculi): it does not recheck c_1 when substituting v' into c_2. Hinze et al. [2006] choose instead to be *picky*, substituting $\langle c_1\rangle^{l',l}\, v'$ into c_2 because it makes their conjunction contract idempotent. We use Findler and Felleisen's rule because it is more familiar; however, both systems are interesting, and we can show (straightforwardly) that both enjoy standard progress and preservation properties. We leave it to future work to see how our full story including the translations applies to the picky system.

We make a standard assumption about constant denotations being well typed: if $\Gamma \vdash k\, v \;:\; T$ then $\Gamma \vdash [\![k]\!](v) \;:\; T$.

4.1 Theorem [Progress]: If $\emptyset \vdash t \;:\; T$ then either $t \longrightarrow_c t'$ or $t = r$ (i.e., $t = v$ or $t = \Uparrow l$).

Proof: By induction on the given derivation.

T_VarT: Contradictory: it is not that case $\emptyset \vdash x \;:\; T$.

T_VarC: Contradictory: it is not that case $\emptyset \vdash x \;:\; \lfloor c \rfloor$.

T_Const: $\emptyset \vdash k \;:\; \mathrm{ty_c}(k)$ and k is a value.

T_Lam: $\emptyset \vdash \lambda x{:}T_1.\; t_2 \;:\; T_1 \to T_2$ and $t = \lambda x{:}T_1.\; t_2$ is a value.

Types and contracts.

$$\begin{array}{rcl} T & ::= & B \mid T_1 \to T_2 \\ c & ::= & \{x{:}B \mid t\} \mid x{:}c_1 \mapsto c_2 \end{array}$$

Terms, values, results, and evaluation contexts.

$$\begin{array}{rcl} t & ::= & x \mid k \mid \lambda x{:}T_1.\ t_2 \mid t_1\, t_2 \mid \\ & & \Uparrow l \mid \langle c\rangle^{l,l'} \mid \langle \{x{:}B \mid t_1\}, t_2, k\rangle^{l} \\ v & ::= & k \mid \lambda x{:}T_1.\ t_2 \mid \langle c\rangle^{l,l'} \mid \langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\, v \\ r & ::= & v \mid \Uparrow l \\ E & ::= & [\,]\ t \mid v\ [\,] \mid \langle \{x{:}B \mid t\}, [\,], k\rangle^{l} \end{array}$$

Figure 9: Syntax of dependent λ_C

$\boxed{t_1 \longrightarrow_c t_2}$ t_1 steps to t_2

$$\frac{}{k\ v \longrightarrow_c [\![k]\!](v)} \quad \text{E\_Const}$$

$$\frac{}{(\lambda x{:}T_1.\ t_2)\ v \longrightarrow_c t_2\{x := v\}} \quad \text{E\_Beta}$$

$$\frac{}{\langle\{x{:}B \mid t\}\rangle^{l,l'}\ k \longrightarrow_c \langle\{x{:}B \mid t\}, t\{x := k\}, k\rangle^{l}} \quad \text{E\_CCheck}$$

$$\frac{}{\langle\{x{:}B \mid t\}, \mathsf{true}, k\rangle^{l} \longrightarrow_c k} \quad \text{E\_OK}$$

$$\frac{}{\langle\{x{:}B \mid t\}, \mathsf{false}, k\rangle^{l} \longrightarrow_c \Uparrow l} \quad \text{E\_Fail}$$

$$\frac{}{(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\ v)\ v' \longrightarrow_c \langle c_2\{x := v'\}\rangle^{l,l'}\ (v\ (\langle c_1\rangle^{l',l}\ v'))} \quad \text{E\_CDecomp}$$

$$\frac{t_1 \longrightarrow_c t_2}{E\,[t_1] \longrightarrow_c E\,[t_2]} \quad \text{E\_Compat}$$

$$\frac{}{E\,[\Uparrow l] \longrightarrow_c \Uparrow l} \quad \text{E\_Blame}$$

Figure 10: Operational semantics for dependent λ_C

Contract erasure.

$$\begin{array}{rcl} \lfloor \{x{:}B \mid t\} \rfloor & = & B \\ \lfloor x{:}c_1 \mapsto c_2 \rfloor & = & \lfloor c_1 \rfloor \to \lfloor c_2 \rfloor \end{array}$$

Typing contexts.

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

Figure 11: Typing rules for dependent λ_C (part 1)

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset} \quad \text{T\_Empty}$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, x{:}T} \quad \text{T\_ExtVarT}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash_c c : \lfloor c \rfloor}{\vdash \Gamma, x{:}c} \quad \text{T\_ExtVarC}$$

$\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T\_VarT}$$

$$\frac{x{:}c \in \Gamma}{\Gamma \vdash x : \lfloor c \rfloor} \quad \text{T\_VarC}$$

$$\frac{}{\Gamma \vdash k : \mathrm{ty_c}(k)} \quad \text{T\_Const}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2} \quad \text{T\_Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \quad \text{T\_App}$$

$$\frac{}{\Gamma \vdash \Uparrow l : T} \quad \text{T\_Blame}$$

$$\frac{\Gamma \vdash_c c : T}{\Gamma \vdash \langle c \rangle^{l,l'} : T \to T} \quad \text{T\_Contract}$$

$$\frac{\begin{array}{c}\emptyset \vdash k : B \qquad \emptyset \vdash t_2 : \mathsf{Bool} \\ \emptyset \vdash_c \{x{:}B \mid t_1\} : B \qquad \vdash t_2 \supset t_1\{x := k\}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B} \quad \text{T\_Checking}$$

$\boxed{\Gamma \vdash_c c : T}$

$$\frac{\Gamma, x{:}B \vdash t : \mathsf{Bool}}{\Gamma \vdash_c \{x{:}B \mid t\} : B} \quad \text{T\_RefineC}$$

$$\frac{\Gamma \vdash_c c_1 : T_1 \qquad \Gamma, x{:}c_1 \vdash_c c_2 : T_2}{\Gamma \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2} \quad \text{T\_FunC}$$

$\boxed{\vdash t_1 \supset t_2}$

$$\frac{t_1 \longrightarrow_c^* \mathsf{true} \quad \text{implies} \quad t_2 \longrightarrow_c^* \mathsf{true}}{\vdash t_1 \supset t_2} \quad \text{T\_Imp}$$

Figure 12: Typing rules for dependent λ_{C} (part 2)

T_App: $\emptyset \vdash t_1\, t_2 \;:\; T_2$, and by inversion $\emptyset \vdash t_1 \;:\; T_1 \rightarrow T_2$ and $\emptyset \vdash t_2 \;:\; T_1$. We show that t takes a step. There are several possibilities:

1. $t_1 \longrightarrow_c t_1'$. Then $t_1\, t_2 \longrightarrow_c t_1'\, t_2$ by E_Compat.
2. $t_1 = v$. Again, there are several possiblities:
   (a) $t_2 \longrightarrow_c t_2'$. Then $v\, t_2 \longrightarrow_c v\, t_2'$ by E_Compat.
   (b) $t_2 = v'$. We proceed by case analysis on v:
      - $v = k$. By inversion, $\emptyset \vdash k \;:\; B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_n$ (recalling the assumption that constants have first-order types) and $\emptyset \vdash v' \;:\; B_1$. The latter means $v' = k' \in \mathcal{K}_{B_1}$, so $[\![k]\!](k')$ is defined and $k\, k' \longrightarrow_c [\![k]\!](k')$ by E_Const.
      - $v = \lambda x{:}T_1.\ t_{12}$. By E_Beta, $(\lambda x{:}T_1.\ t_{12})\, v' \longrightarrow_c t_{12}\{x := v'\}$.
      - $v = \langle c\rangle^{l,l'}$. If $c = \{x{:}B \mid t_0\}$, then by inversion $\emptyset \vdash v' \;:\; B$, so $v' = k \in \mathcal{K}_B$. We have $\langle\{x{:}B \mid t_0\}\rangle^{l,l'}\, k \longrightarrow_c \langle\{x{:}B \mid t_0\}, t_0\{x := k\}, k\rangle^l$ by E_CCheck. If $c = x{:}c_1 \mapsto c_2$, then $\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\, v'$ is a value.
      - $v = \langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\, v_0$. We have $(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\, v_0)\, v' \longrightarrow_c \langle c_2\{x := v'\}\rangle^{l,l'}\, (v_0\, (\langle c_1\rangle^{l',l}\, v'))$ by E_CDecomp.
   (c) $t_2 = \Uparrow l$. Then $t_1 \Uparrow l \longrightarrow_c \Uparrow l$ by E_Blame.
3. $t_1 = \Uparrow l$. Then $\Uparrow l\, t_2 \longrightarrow_c \Uparrow l$ by E_Blame.

T_Blame: $\emptyset \vdash \Uparrow l \;:\; T$ and $t = \Uparrow l$, which is a result.

T_Contract: $\emptyset \vdash \langle c\rangle^{l,l'} \;:\; T \rightarrow T$, and $t = \langle c\rangle^{l,l'}$, which is a value.

T_Checking: $\emptyset \vdash \langle\{x{:}B \mid t_1\}, t_2, k\rangle^l \;:\; B$. We show that t takes a step. There are three possibilities:

1. $t_2 \longrightarrow_c t_2'$. Then $\langle\{x{:}B \mid t_1\}, t_2, k\rangle^l \longrightarrow_c \langle\{x{:}B \mid t_1\}, t_2', k\rangle^l$ by E_Compat.
2. $t_2 = v$. By inversion, $\emptyset \vdash v \;:\; \mathsf{Bool}$, so v is either true or false. If $v = \mathsf{true}$, then E_OK applies and $t' = k$. If $v = \mathsf{false}$, then E_Fail applies and $t' = \Uparrow l$. Either way, t takes a step.
3. $t_2 = \Uparrow l''$. In this case, $\langle\{x{:}B \mid t_1\}, \Uparrow l'', k\rangle^l \longrightarrow_c \Uparrow l''$ by E_Blame. □

For preservation, we reprove the familiar confluence lemma, along with weakening and substitution. Note that our substitution lemma must now also cover contracts, since they are no longer closed.

4.2 Lemma [Determinacy]: If $t \longrightarrow_c t'$ and $t \longrightarrow_c t''$, then $t' = t'$

Proof: By induction on the first derivation, recalling in the E_Const case that $[\![-]\!]$ is a function. □

4.3 Corollary [Coevaluation]: If $t \longrightarrow_c^* r$ and $t \longrightarrow_c^* t'$, then $t' \longrightarrow_c^* r$.

4.4 Lemma [Weakening]: If $\Gamma \vdash t \;:\; T$ and $dom(\Gamma) \cap dom(\Gamma') = \emptyset$ then $\Gamma\,,\, \Gamma' \vdash t \;:\; T$.

Proof: Straightforward induction on t. □

4.5 Lemma [Term and contract substitution]: If $\emptyset \vdash v \;:\; T'$, then

1. if $\Gamma, x{:}T'\,,\, \Gamma' \vdash t \;:\; T$ then $\Gamma\,,\, \Gamma' \vdash t\{x := v\} \;:\; T$, and
2. if $\Gamma, x{:}T'\,,\, \Gamma' \vdash_c c \;:\; T$ then $\Gamma\,,\, \Gamma' \vdash_c c\{x := v\} \;:\; T$.

Proof: By mutual induction on the typing derivations for t and c.

T_VarT: We have $\Gamma, x{:}T'\,, \Gamma' \vdash y : T$. If $x = y$, then $y\{x := v\} = v$ and $T' = T$. Given $\emptyset \vdash v : T'$, we have $\Gamma\,, \Gamma' \vdash v : T'$ by weakening (Lemma 4.4).

On the other hand, if $x \neq y$, then y was in Γ or Γ', so we have $\Gamma\,, \Gamma' \vdash y : T$.

T_Const: Immediate.

T_Lam: We have $\Gamma, x{:}T'\,, \Gamma' \vdash \lambda y{:}T_1.\ t_2 : T_1 \to T_2$.

$\Gamma\,, \Gamma', y{:}T_1 \vdash t_2\{x := v\} : T_2$ by the IH. Then $\Gamma\,, \Gamma' \vdash \lambda y{:}T_1.\ (t_2\{x := v\}) : T_1 \to T_2$ by T_Lam, and $\Gamma\,, \Gamma' \vdash (\lambda y{:}T_1.\ t_2)\{x := v\} : T_1 \to T_2$ by the definition of substitution.

T_App: By IH.

T_Blame: Immediate.

T_Contract: By IH.

T_Checking: Contradictory—only applies in the empty context.

T_RefineC: By IH.

T_FunC: By IH. □

4.6 Theorem [Preservation]: If $\emptyset \vdash t : T$ and $t \longrightarrow_c t'$ then $\emptyset \vdash t' : T$.

Proof: By induction on the typing derivation.

T_VarT: Contradictory—it is not possible that $\emptyset \vdash x : T$ for any T.

T_VarC: Contradictory—it is not possible that $\emptyset \vdash x : \lfloor c \rfloor$ for any c.

T_Const: Contradictory—it is not possible that $k \longrightarrow_c t'$ for any t'.

T_Lam: Contradictory—it is not possible that $\lambda x{:}T_1.\ t_2 \longrightarrow_c t'$ for any t'.

T_App: $\emptyset \vdash t_1\, t_2 : T_2$; by inversion, $\emptyset \vdash t_1 : T_1 \to T_2$ and $\emptyset \vdash t_2 : T_1$. Six rules might have been used to derive $t_1\, t_2 \longrightarrow_c t'$:

E_Const: $k\, v \longrightarrow_c [\![k]\!](v)$. Since constants are first-order, $\emptyset \vdash k : B_1 \to B_2 \to \ldots \to B_n$ and $\emptyset \vdash v : B_1$. We assume that type assignment and denotations are consistent, so if $\emptyset \vdash k\, v : T$, then $\emptyset \vdash [\![k]\!](v) : T$.

E_Beta: We have $\emptyset \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2$ and $\emptyset \vdash v : T_1$. We must show that $\emptyset \vdash t_2\{x := v\} : T_2$.

By inversion of the typing of the lambda, we know that $\emptyset, x{:}T_1 \vdash t_2 : T_2$. The substitution lemma (Lemma 4.5) completes this case.

E_CDecomp: We know $(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\, v)\, v' \longrightarrow_c \langle c_2\{x := v'\}\rangle^{l,l'}\, (v\, (\langle c_1\rangle^{l',l}\, v'))$, and by inversion $\emptyset \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2$, as well as $\emptyset \vdash v : T_1 \to T_2$ and $\emptyset \vdash v' : T_1$.

By further inversion, $\emptyset \vdash_c c_1 : T_1$ and $x{:}T_1 \vdash_c c_2 : T_2$. These assumptions, along with T_App, T_Contract, and substitution for contracts (Lemma 4.5) allow us to show that $\emptyset \vdash \langle c_2\{x := v'\}\rangle^{l,l'}\, (v\, (\langle c_1\rangle^{l',l}\, v')) : T_2$.

E_CCheck: We know that $\emptyset \vdash \langle\{x{:}B \mid t\}\rangle^{l,l'}\, k : T$. We must show that $\emptyset \vdash \langle\{x{:}B \mid t\}, t\{x := k\}, k\rangle^{l} : T$. By inversion of the typing derivation, we know that $\emptyset \vdash k : T$ and $\emptyset \vdash_c \{x{:}B \mid t\} : T$ (where $\mathrm{ty_c}(k) = T = B!$). Inversion of the latter derivation then shows that $x{:}B \vdash t : \mathsf{Bool}$.

By the substitution lemma (Lemma 4.5), we know that $\emptyset \vdash t\{x := k\} : \mathsf{Bool}$. $\vdash t\{x := k\} \supset t\{x := k\}$ is immediate. We can now use T_Checking to show that $\emptyset \vdash \langle\{x{:}B \mid t\}, t\{x := k\}, k\rangle^{l} : B$, and we are done.

E_Compat: In this case, $t_1\, t_2 = E\,[t'']$ and $t'' \longrightarrow_c t'''$. By inversion, $\emptyset \vdash t_1 : T_1 \to T_2$ and $\emptyset \vdash t_2 : T_1$. There are two possible contexts E.

If $E = [\,]\ t_2$, then $t_1 \longrightarrow_c t_1'$. By the IH, $\emptyset \vdash t_1' : T_1 \to T_2$. By T_App, $\emptyset \vdash t_1'\, t_2 : T_2$.

If $E = v_1\ [\,]$ (where $v_1 = t_1$), then $t_2 \longrightarrow_c t_2'$. By the IH, $\emptyset \vdash t_2' : T_1$. By T_App, $\emptyset \vdash v_1\, t_2' : T_2$.

E_Blame: By T_Blame, $\emptyset \vdash \Uparrow l \ : \ T_2$.

T_Blame: Contradictory—it is not possible that $\Uparrow l \ \longrightarrow_c \ t'$ for any t'.

T_Contract: Contradictory—$\langle c \rangle^{l,l'}$ is a value.

T_Checking: $\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^l \ : \ B$. Four evaluation rules could have been used to derive $\langle \{x{:}B \mid t_1\}, t_2, k \rangle^l \ \longrightarrow_c \ t'$:

E_OK: $t' = k$. We see $\emptyset \vdash k \ : \ B$ by inversion of the typing derivation.

E_Fail: Here $t' = \Uparrow l$; by T_Blame, $\emptyset \vdash \Uparrow l \ : \ B$.

E_Compat: The evaluation context is of the form $E = \langle \{x{:}B \mid t_1\}, [\,], k \rangle^l$. We know that $\emptyset \vdash E\,[t_2] \ : \ B$, that $t_2 \ \longrightarrow_c \ t_2'$, and that $\vdash t_2 \supset t_1\{x := k\}$. By inversion of the typing derivation, $\emptyset \vdash t_2 \ : \ \mathsf{Bool}$. By the IH, $\emptyset \vdash t_2' \ : \ \mathsf{Bool}$. By coevaluation (Corollary 4.3), we find that $\vdash t_2' \supset t_1\{x := k\}$, so we can conclude that $\emptyset \vdash E\,[t_2'] \ : \ B$.

E_Blame: Here $t' = \Uparrow l'$. By T_Blame, $\emptyset \vdash \Uparrow l' \ : \ B$. □

Types

$$S \quad ::= \quad \{x{:}B \mid s\} \;\mid\; x{:}S_1 \to S_2$$

Operational semantics

$$\frac{}{\begin{array}{l}(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\, w)\; w' \longrightarrow_h \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow S_{22}\{x := w'\}\rangle^l \\ \quad (w\;(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w'))\end{array}} \quad \text{F\_CDecomp}$$

Typing rules

$$\frac{\Delta \vdash s_1 : (x{:}S_1 \to S_2) \qquad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\, s_2 : S_2\{x := s_2\}} \quad \text{S\_App}$$

$$\frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\Delta \vdash \{x{:}B \mid s\}} \quad \text{WF\_Refine}$$

$$\frac{\Delta \vdash S_1 \qquad \Delta, x{:}S_1 \vdash S_2}{\Delta \vdash x{:}S_1 \to S_2} \quad \text{WF\_Fun}$$

$$\frac{\Delta \vdash S_{21} <: S_{11} \qquad \Delta, x{:}S_{21} \vdash S_{12} <: S_{22}}{\Delta \vdash x{:}S_{11} \to S_{12} <: x{:}S_{21} \to S_{22}} \quad \text{Sub\_Fun}$$

$$\frac{\forall \sigma.\ ((\Delta \models \sigma \,\text{and}\, \sigma(s_1) \longrightarrow_h^* \mathsf{true}) \ \text{ implies } \ \sigma(s_2) \longrightarrow_h^* \mathsf{true})}{\Delta \vdash s_1 \supset s_2} \quad \text{S\_Imp}$$

Closing Substitutions

$$\frac{}{\emptyset \models \emptyset} \quad \text{CS\_Empty}$$

$$\frac{s \in [\![S]\!] \qquad \Delta\{x := s\} \models \sigma}{x{:}S, \Delta \models \sigma\{x := s\}} \quad \text{CS\_Ext}$$

Figure 13: Changes for dependent λ_{H}

Dependent $\lambda_{\mathbf{H}}$

Now we come to the challenging part: dependent λ_{H} and its proof of type soundness.[4] These results require the most complex metatheory in the paper because we need some strong properties about call-by-value evaluation. However, the benefit of a CBV semantics is a better treatment of blame. By contrast, Knowles and Flanagan [2009] cannot treat failed casts as exceptions because that would destroy confluence. The

[4]The proof of type soundness for this system is significantly different from the soundness proof in Knowles and Flanagan [2009], where the operational semantics of λ_{H} is full, nondeterministic β-reduction. At first glance, it might seem that our preservation and progress theorems follow directly from the results for Knowles and Flanagan's language, since CBV is a restriction of full β-reduction. However, note that the reduction relation is used in the type system (in rule S_Imp), so the type systems for the two languages are not the same. For example, suppose that the term *bad* contains a cast that fails. In our system the type $\{y{:}B \mid \mathsf{true}\}$ is not a subtype of $\{y{:}B \mid (\lambda x{:}S.\ \mathsf{true})\ \mathit{bad}\}$ because the contract evaluates to blame. However, the subtyping does hold in the Knowles and Flanagan system because in that language the predicate reduces to true.

Types.

$$S \quad ::= \quad \{x{:}B \mid s_1\} \;\mid\; x{:}S_1 \to S_2$$

Terms, values, results, and evaluation contexts.

$$\begin{array}{rcl} s & ::= & x \mid k \mid \lambda x{:}S_1.\ s_2 \mid s_1\, s_2 \mid \\ & & \Uparrow l \mid \langle S_1 \Rightarrow S_2\rangle^l \mid \langle\{x{:}B \mid s_1\}, s_2, k\rangle^l \\ w & ::= & k \mid \lambda x{:}S_1.\ s_2 \mid \langle S_1 \Rightarrow S_2\rangle^l \mid \\ & & \langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\, w \\ q & ::= & w \mid \Uparrow l \\ F & ::= & [\,]\ s \mid w\ [\,] \mid \langle\{x{:}B \mid s\}, [\,], k\rangle^l \end{array}$$

Figure 14: Syntax of dependent λ_{H}

$\boxed{s_1 \rightsquigarrow_h s_2}$ s_1 reduces to s_2

$$\frac{}{k\, w \rightsquigarrow_h [\![k]\!](w)} \quad \textsc{F\_Const}$$

$$\frac{}{(\lambda x{:}S.\ s_{12})\, w_2 \rightsquigarrow_h s_{12}\{x := w_2\}} \quad \textsc{F\_Beta}$$

$$\frac{}{\langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\, k \rightsquigarrow_h \langle\{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l} \quad \textsc{F\_CCheck}$$

$$\frac{}{\langle\{x{:}B \mid s\}, \mathsf{true}, k\rangle^l \rightsquigarrow_h k} \quad \textsc{F\_OK}$$

$$\frac{}{\langle\{x{:}B \mid s\}, \mathsf{false}, k\rangle^l \rightsquigarrow_h \Uparrow l} \quad \textsc{F\_Fail}$$

$$\frac{}{(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\, w)\, w' \rightsquigarrow_h \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow S_{22}\{x := w'\}\rangle^l\, (w\, (\langle S_{21} \Rightarrow S_{11}\rangle^l\, w'))} \quad \textsc{F\_CDecomp}$$

$\boxed{s_1 \longrightarrow_h s_2}$ s_1 steps to s_2

$$\frac{s_1 \rightsquigarrow_h s_2}{s_1 \longrightarrow_h s_2} \quad \textsc{F\_Reduce}$$

$$\frac{s_1 \longrightarrow_h s_2}{F\,[s_1] \longrightarrow_h F\,[s_2]} \quad \textsc{F\_Compat}$$

$$\frac{}{F\,[\Uparrow l] \longrightarrow_h \Uparrow l} \quad \textsc{F\_Blame}$$

Figure 15: Operational semantics for dependent λ_{H}

needed extensions are detailed in Figures 13 and 16.[5] The full definitions are in Figures 17 and 18.

First, we enrich the type system with dependent function types, $x{:}S_1 \to S_2$, where x may appear in

[5]The semantics in these figures is the same as that of Flanagan [2006] except for the evaluation relation, the treatment of blame as an exception, and a change to the type semantics that we discuss below.

Denotations of types and kinds

$$
\begin{array}{rcl}
s \in [\![\{x{:}B \mid s_0\}]\!] & \iff & s \longrightarrow_h^* \Uparrow l \\
& & \vee\ \ (\exists k \in \mathcal{K}_B.\ s \longrightarrow_h^* k \\
& & \quad \wedge\ s_0\{x := k\} \longrightarrow_h^* \mathsf{true}) \\
s \in [\![x{:}S_1 \to S_2]\!] & \iff & \forall q \in [\![S_1]\!].\ s\ q \in [\![S_2\{x := q\}]\!] \\[1em]
\{x{:}B \mid s\} \in [\![\star]\!] & \iff & \forall k \in \mathcal{K}_B. \\
& & \quad s\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!] \\
x{:}S_1 \to S_2 \in [\![\star]\!] & \iff & S_1 \in [\![\star]\!] \\
& & \wedge\ \ \forall q \in [\![S_1]\!].\ S_2\{x := q\} \in [\![\star]\!]
\end{array}
$$

Semantic judgments

$$
\begin{array}{rcl}
& & \forall \sigma \text{ s.t. } \Delta \models \sigma : \\
\Delta \models S_1 <: S_2 & \iff & \quad [\![\sigma(S_1)]\!] \subseteq [\![\sigma(S_2)]\!] \\
\Delta \models s : S & \iff & \quad \sigma(s) \in [\![\sigma(S)]\!] \\
\Delta \models S & \iff & \quad \sigma(S) \in [\![\star]\!]
\end{array}
$$

Figure 16: Type and kind semantics for dependent λ_{H}

$$
\begin{array}{rclcrcl}
\lfloor \{x{:}B \mid s\} \rfloor & = & B & \quad & \lfloor S_1 \to S_2 \rfloor & = & \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor \\
\lfloor \{x{:}B \mid t\} \rfloor & = & B & & \lfloor c_1 \mapsto c_2 \rfloor & = & \lfloor c_1 \rfloor \to \lfloor c_2 \rfloor \\
\lfloor T \rfloor & = & T & & & &
\end{array}
$$

Figure 17: Type erasure for dependent λ_{H}

S_2. This dependency must be maintained through higher-order casts in the rule F_CDecomp. As the cast decomposes, the variables in the codomain types of such a cast must be replaced by the argument. However, this substitution is asymmetric; on one side, we cast that argument and on the other we do not. This behavior is required for type preservation. For suppose we have $\Delta \vdash x{:}S_{11} \to S_{12}$ and $\Delta \vdash x{:}S_{21} \to S_{22}$ with equal skeletons, and values $\Delta \vdash w : (x{:}S_{11} \to S_{12})$ and $\Delta \vdash w' : S_{21}$. Then $\Delta \vdash (\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\, w)\, w' : S_{22}\{x := w'\}$. To preserve variable scoping when we decompose the cast, we must make *some* substitution into S_{12} and S_{22}, but which? It is clear that we must substitute w' into S_{22}, since the original application has type $S_{22}\{x := w'\}$. Decomposing the cast will produce the inner application $\Delta \vdash w\,(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w') : S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\}$. Thus, to be able to apply the codomain cast, we must substitute $\langle S_{21} \Rightarrow S_{11}\rangle^l\, w'$ into S_{12}. This calculation tells us exactly how F_CDecomp has to look.

While the operational semantics change only in F_CDecomp, we've split the evaluation relation into two parts: reductions $\rightsquigarrow_h$ and steps $\longrightarrow_h$. This is a technical change that allows us to factor our proofs more cleanly (particularly for the parallel reduction proofs).

Next, we change the typing rules to admit dependency. The new application rule, S_App, substitutes the argument for the variable in the result of an application. We generalize the refinement-type formation rule, WF_Refine, to allow predicates that use variables from the enclosing context. The formation rule for function types adds the bound variable to the context when checking the codomain. In Sub_Fun, subtyping for dependent function types remains contravariant, but we also add the argument variable to the context with the smaller type. This is similar to the function subtyping rule of $F_{<:}$ [Cardelli et al., 1991].

The final change is that, in S_Imp, the terms s_1 and s_2 may mention variables in the context. Therefore, before we can compare their evaluation behavior, we must first quantify over all closing substitutions σ satisfying Δ (written $\Delta \models \sigma$).

Some care is needed here to prevent the typing rules from becoming circular: the typing rule S_Sub references the subtyping judgment. The subtyping rule Sub_Refine references the implication judgment. The single implication rule S_Imp has $\Delta \models \sigma$ in a negative position. To guarantee the well-definedness of the type system, we must not allow $\Delta \models \sigma$ to refer back to the other judgments. (The reader may wonder why this wasn't a problem with the T_Imp rule in λ_C, but notice that T_Imp has no context. If we only needed S_Imp for the S_Checking rule, we wouldn't need contexts here, either—we can ensure that active checks only occur at the top-level, with an empty context. But the Sub_Refine subtyping rule refers to S_Imp, and this may be used in arbitrary contexts.)

To avoid circularity, we define a *denotational semantics* for λ_H's types.[6] The basic idea is that the semantics of a type is a set of closed terms that is defined independently of the syntactic typing relation, but that turns out to contain all closed well-typed terms of that type. Thus, in the definition of $\Delta \models \sigma$, we quantify over a somewhat larger set than necessary—not just the syntactically well-typed terms of appropriate type (which are all the ones that will ever appear in programs), but all semantically well-typed ones.

The type semantics appears in Figure 16. It is defined by induction on the structure of type skeletons. For refinement types, terms must either go to blame or produce a constant that satisfies (all instances of) the given predicate. For function types, well-typed arguments must go to well-typed results. (Notice that, by construction, these sets include only terminating terms that do not get stuck.)

We must again make the assumption that constants have most-specific types: if $\lfloor \mathrm{ty_h}(k) \rfloor = B$ and $s\{x := k\} \longrightarrow_h^* \mathsf{true}$ then $\emptyset \vdash \mathrm{ty_h}(k) <: \{x{:}B \mid s\}$. We make some other, more standard assumptions, as well. Constants must have closed, well-formed types, and the types assigned must be well-formed.We note that this last is true by definition at base types, but must be assumed at higher-order types.

We introduce a few facts about the type semantics before proving semantic type soundness.

4.7 Lemma [Determinacy]: If $s \longrightarrow_h s'$ and $s \longrightarrow_h s''$, then $s' = s'$

Proof: By induction on the first derivation, recalling in the F_Const case that $[\![-]\!]$ is a function. □

4.8 Corollary [Coevaluation]: If $s \longrightarrow_h^* s'$ and $s \longrightarrow_h^* q$, then $s' \longrightarrow_h^* q$.

4.9 Lemma [Expansion and contraction of $[\![S]\!]$]: If $s \longrightarrow_h^* s'$, then $s' \in [\![S]\!]$ iff $s \in [\![S]\!]$.

Proof: By induction on $|S|$.

- $S = \{x{:}B \mid s_1\}$. By coevaluation (Corollary 4.8), s and s' go to the same result, $q \in [\![S]\!]$.
- $S = x{:}S_1 \to S_2$. Let $q \in [\![S_1]\!]$. We know that $s\,q \in [\![S_2\{x := q\}]\!]$, and $s\,q \longrightarrow_h^* s'\,q$. By the IH, $s'\,q \in [\![S_2\{x := q\}]\!]$. The other direction is similar. □

4.10 Lemma [Blame inhabits all types]: For all S, $\Uparrow l \in [\![S]\!]$.

Proof: By induction on $|S|$.

- $S = \{x{:}B \mid s\}$. Since $\Uparrow l \longrightarrow_h^* \Uparrow l$, we have $\Uparrow l \in [\![\{x{:}B \mid s\}]\!]$ by definition.
- $S = x{:}S_1 \to S_2$. Let $q \in [\![S_1]\!]$; we show $\Uparrow l\, q \in [\![S_2\{x := q\}]\!]$. By F_Blame $\Uparrow l\, q \longrightarrow_h \Uparrow l$, and $\Uparrow l \in [\![S_2\{x := q\}]\!]$ by the IH. We conclude by expansion (Lemma 4.9).

□

We require that constants are semantically well typed: $k \in [\![\mathrm{ty_h}(k)]\!]$.

4.11 Corollary [Nonemptiness]: For all S, there exists some q such that $q \in [\![S]\!]$.

The normal forms of $\longrightarrow_h^*$ are of the form $q = w$ or $\Uparrow l$.

[6]Knowles and Flanagan [2009] also introduce a type semantics, but it differs from ours in two ways. First, because they cannot treat blame as an exception (because their semantics is nondeterministic) they must restrict the terms in the semantics to be those that only get stuck at failed casts. They do so by requiring the terms to be well-typed in the simply-typed lambda calculus after all casts have been erased. Secondly, their type semantics does not require strong normalization. However, it is not clear whether their language actually admits nontermination—they include a fix constant, but their semantic type soundness proof appears to break down in that case.

4.12 Lemma [Strong normalization]: If $s \in [\![S]\!]$, then there exists a q such that $s \longrightarrow_h^* q$, i.e., either $s \longrightarrow_h^* w$ or $s \longrightarrow_h^* \Uparrow l$.

Proof: By induction on $|S|$.

- $S = \{x{:}B \mid s_0\}$. Suppose $s \in [\![\{x{:}B \mid s_0\}]\!]$. By definition, either $s \longrightarrow_h^* w$ or $s \longrightarrow_h^* \Uparrow l$, so s normalizes.
- $S = x{:}S_1 \to S_2$. Suppose $s \in [\![x{:}S_1 \to S_2]\!]$. We know that for any $q \in [\![S_1]\!]$ that $s\,q \in [\![S_2\{x := q\}]\!]$. Since $[\![S_1]\!]$ is nonempty (by Lemma 4.11), let $q \in [\![S_1]\!]$. By the IH, $s\,q \longrightarrow_h^* w$ or $s\,q \longrightarrow_h^* \Uparrow l$. By the definition of evaluation contexts and $\longrightarrow_h^*$, the function position is evaluated first. So if $s\,q \longrightarrow_h^* w$, we must first have $s\,q \longrightarrow_h^* w'\,q$, and so $s \longrightarrow_h^* w'$; alternatively, we could have $s\,q \longrightarrow_h^* \Uparrow l$. There are two ways for this to happen: either $s \longrightarrow_h^* \Uparrow l$, or $s \longrightarrow_h^* w'$ and $q \longrightarrow_h^* \Uparrow l$. In any case, s normalizes. □

What we want to know about the type semantics is *semantic type soundness*: if $\emptyset \vdash s : S$, then $s \in [\![S]\!]$. However, to prove this, we must generalize it. In the bottom of Figure 16, we define three *semantic judgements* that correspond to each of the three typing judgments. (Note that the third one requires the definition of a *kind* semantics that picks out well-behaved types—those whose embedded contracts belong to the type semantics.) We then show that the typing judgments imply their semantic counterparts.

4.13 Theorem [Semantic type soundness]:

1. If $\Delta \vdash S_1 <: S_2$ then $\Delta \models S_1 <: S_2$
2. If $\Delta \vdash s : S$ then $\Delta \models s : S$
3. If $\Delta \vdash S$ then $\Delta \models S$

Proof: Proof of (1) is in Lemma 4.14. Proof of (2) and (3) is in Lemma 4.21. □

The first part follows by induction on the subtyping judgment.

4.14 Lemma [Semantic subtype soundness]: If $\Delta \vdash S_1 <: S_2$ then $\Delta \models S_1 <: S_2$.

Proof: By induction on the subtyping derivation.

<u>Sub_Refine</u>: We know $\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}$, and must show the corresponding semantic subtyping. Inversion of this derivation gives us $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 \supset s_2$, which means:

$$\forall \sigma.\ ((\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \models \sigma \text{ and } \sigma(s_1) \longrightarrow_h^* \mathsf{true}) \quad \text{implies} \quad \sigma(s_2) \longrightarrow_h^* \mathsf{true}) \qquad (*)$$

We must show $\Delta \models \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}$, i.e., that $\forall \sigma.\ (\Delta \models \sigma \quad \text{implies} \quad [\![\{x{:}B \mid s_1\}]\!] \subseteq [\![\{x{:}B \mid s_2\}]\!])$. Let σ be given such that $\Delta \models \sigma$. Suppose $s \in [\![\sigma(\{x{:}B \mid s_1\})]\!]$. By definition, either s goes to $\Uparrow l$, or it goes to $k \in \mathcal{K}_B$ such that $s_1\{x := k\} \longrightarrow_h^* \mathsf{true}$. In the former case, $\Uparrow l \in [\![\{x{:}B \mid s_2\}]\!]$ by definition. So consider the latter case, where $s \longrightarrow_h^* k$.

We already know that $k \in \mathcal{K}_B$, so it remains to see that:

$$\sigma(s_2)\{x := k\} \longrightarrow_h^* \mathsf{true}$$

We know by assumption that $\sigma(s_1)\{x := k\} \longrightarrow_h^* \mathsf{true}$. By Lemma 4.27, $k \in [\![\{x{:}B \mid \mathsf{true}\}]\!]$.

Now observe that $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \models \sigma\{x := k\}$. Since $\sigma'(s_1) \longrightarrow_h^* \mathsf{true}$, we can conclude that $\sigma'(s_2) \longrightarrow_h^* \mathsf{true}$ by our assumption (*). This completes this case.

<u>Sub_Fun</u>: $\Delta \vdash (x{:}S_{11} \to S_{12}) <: (x{:}S_{21} \to S_{22})$; by the IH, we have $\Delta \models S_{21} <: S_{11}$ and $\Delta, x{:}S_{21} \models S_{12} <: S_{22}$. We must show that $\Delta \models (x{:}S_{11} \to S_{12}) <: (x{:}S_{21} \to S_{22})$.

Let $\Delta \models \sigma$ and $s \in [\![\sigma(x{:}S_{11} \to S_{12})]\!]$, for some σ. We must show, for all q, that if $q \in [\![\sigma(S_{21})]\!]$ then $s\,q \in [\![\sigma(S_{22})\{x := q\}]\!]$.

Let $q \in [\![\sigma(S_{21})]\!]$. Then $q \in [\![\sigma(S_{11})]\!]$. Since $s \in [\![\sigma(x{:}S_{11} \to S_{12})]\!]$, we know that $s\,q \in [\![\sigma(S_{12})\{x := q\}]\!]$. Finally, since $\Delta, x{:}S_{21} \models S_{12} <: S_{22}$ and $\Delta, x{:}S_{21} \models \sigma\{x := q\}$, we can conclude that $s\,q \in [\![\sigma(S_{22})\{x := q\}]\!]$, and so $s \in [\![\sigma(x{:}S_{21} \to S_{22})]\!]$. □

However, we run into complications with the second and third parts (which must be proven together). The crux of the difficulty lies with the S_APP rule. Suppose the application $s_1\,s_2$ was well typed and $s_1 \in [\![x{:}S_1 \to S_2]\!]$ and $s_2 \in [\![S_1]\!]$. According to S_APP, the application's type is $S_2\{x := s_2\}$. By the type semantics defined in Figure 16, if $s_1 \in [\![x{:}S_1 \to S_2]\!]$, then $s_1\,q \in [\![S_2\{x := q\}]\!]$ for any $q \in [\![S_1]\!]$. Sadly, s_2 isn't necessarily a result! We do know, however, that $s_2 \in [\![S_1]\!]$, so $s_2 \longrightarrow_h^* q_2$ by strong normalization (Lemma 4.12). We need to ask, then, how the type semantics of $S_2\{x := s_2\}$ and $S_2\{x := q_2\}$ relate. (One might think that we can solve this by changing the type semantics to quantify over terms, not results. But this just pushes the problem to the S_LAM case.)

We can show that the two type semantics are in fact equal using a parallel reduction technique. We define a parallel reduction relation $\Rrightarrow$ on terms and types in Figure 19 that allows redices in different parts of a term (or type) to be reduced in the same step, and we prove that types that parallel-reduce to each other—like $S_2\{x := s_2\}$ and $S_2\{x := q_2\}$—have the same semantics. The definition of parallel reduction is standard except that we need to be careful to make it respect our call-by-value reduction order: the *beta*-redex $(\lambda x{:}S_1.\ s_1)\,s_2$ should not be contracted (though other redices within s_1 and s_2 can be) unless s_2 is a value, since this can change the order of effects.[7]The proof requires a longish sequence of technical lemmas that essentially show that $\Rrightarrow$ commutes with $\longrightarrow_h^*$. A Coq development team [August 2009] development of these properties (using Aydemir and Weirich [March 2009]) is available at `http://www.cis.upenn.edu/~mgree/papers/lambdah_parred.tgz`. We restate the critical results here.

Lemma [Substitution of parallel-reducing terms, Lemma `A3` in `thy.v`]: If $w \Rrightarrow w'$, then

1. if $s \Rrightarrow s'$ then $s\{x := w\} \Rrightarrow s'\{x := w'\}$, and
2. if $S \Rrightarrow S'$ then $S\{x := w\} \Rrightarrow S'\{x := w'\}$.

Lemma [Parallel reduction implies coevaulation, Lemma `A20` in `thy.v`]: If $s_1 \Rrightarrow s_2$ then $s_1 \longrightarrow_h^* k$ iff $s_2 \longrightarrow_h^* k$. Similarly, $s_1 \longrightarrow_h^* \Uparrow l$ iff $s_2 \longrightarrow_h^* \Uparrow l$.

One alternative would be to use $\Rrightarrow$ in the typing rules and $\longrightarrow_h$ in the operational semantics. This could simplify some of our metatheory, but it would complicate the specification of the language. Using $\longrightarrow_h$ in the typing rules gives a clearer intuition, as well, and keeps the core system small.

4.17 Lemma [Single parallel reduction preserves type semantics]: If $S_1 \Rrightarrow S_2$ then $[\![S_1]\!] = [\![S_2]\!]$.

Proof: By induction on $|S_1|$ (which is equal to $|S_2|$), with a case analysis on the final rule used to show $S_1 \Rrightarrow S_2$.

P_SREFL: Here $S_1 = S_2$, and so $[\![S_1]\!] = [\![S_2]\!]$ trivially.

P_SREFINE: $S_1 = \{x{:}B \mid s_1\}$ and $S_2 = \{x{:}B \mid s_2\}$ where $s_1 \Rrightarrow s_2$.

First, suppose $s \in [\![S_1]\!]$. Then either $s \longrightarrow_h^* \Uparrow l$ or $s \longrightarrow_h^* k$ where $s_1\{x := k\} \longrightarrow_h^*$ `true`. In the former case, $s \in [\![S_2]\!]$ by Lemma 4.10. In the latter case, $s_1\{x := k\} \Rrightarrow s_2\{x := k\}$ by Lemma `A3`. By coevaluation (Lemma `A20`) we see that $s_2\{x := k\} \longrightarrow_h^*$ `true`, and so $s \in [\![S_2]\!]$.

The other direction is similar.

P_SFUN: $S_1 = x{:}S_{11} \to S_{12}$ and $S_2 = x{:}S_{21} \to S_{22}$, where $S_{11} \Rrightarrow S_{21}$ and $S_{12} \Rrightarrow S_{22}$.

Suppose $s \in [\![x{:}S_{11} \to S_{12}]\!]$. Then for all $q' \in [\![S_{11}]\!]$, we have $s\,q' \in [\![S_{12}\{x := q'\}]\!]$. By the IH, $q' \in [\![S_{21}]\!]$ (since the two denotations are equal). Similarly, since $S_{12} \Rrightarrow S_{22}$ and $q' \Rrightarrow q'$ by P_REFL, we have $S_{12}\{x := q'\} \Rrightarrow S_{22}\{x := q'\}$ by Lemma `A3`; by the IH, these types have equal denotations, so $s\,q' \in [\![S_{12}\{x := q'\}]\!]$ iff $s\,q' \in [\![S_{22}\{x := q'\}]\!]$. □

4.18 Corollary: [Parallel reduction preserves type semantics] If $S_1 \Rrightarrow^* S_2$ then $[\![S_1]\!] = [\![S_2]\!]$.

4.19 Lemma [Partial semantic substitution]: If

[7]We conjecture that a similar "CBV-respecting" variant of full β-reduction could be used in place of our parallel reduction. It is not clear whether it would lead to shorter proofs.

- $\Delta_1, x{:}S', \Delta_2 \models s : S$,
- $\Delta_1, x{:}S', \Delta_2 \models S$, and
- $\Delta_1 \models s' : S'$

then $\Delta_1, \Delta_2\{x := s'\} \models s\{x := s'\} : S\{x := s'\}$ and $\Delta_1, \Delta_2\{x := s'\} \models S\{x := s'\}$.

Proof: By unfolding CS_Ext into the semantic definitions. □

4.20 Lemma [Semantic typing for casts]: If

- $\Delta \models S_1$,
- $\Delta \models S_2$ and
- $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$

then $\Delta \models \langle S_1 \Rightarrow S_2 \rangle^l : x{:}S_1 \to S_2$ for fresh x.

Proof: By induction on $|S_1| = |S_2|$, going by cases on the shape of S_2. Let $\Delta \models \sigma$; we show that $\sigma(\langle S_1 \Rightarrow S_2 \rangle^l) \in [\![\sigma(S_1 \to S_2)]\!]$.

- $S_2 = \{x{:}B \mid \sigma(s_2)\}$. Let $q \in [\![\sigma(S_1)]\!]$. If $q = \Uparrow l'$, then the applied cast goes to $\Uparrow l'$, and we are done by Lemma 4.10. So $q = k \in \mathcal{K}_B$. By F_CCheck $\langle S_1 \Rightarrow \{x{:}B \mid \sigma(s_2)\}\rangle^l\, k \longrightarrow_h \langle \{x{:}B \mid \sigma(s_2)\}, \sigma(s_2)\{x := k\}, k\rangle^l$. By the well-kinding of S_2, we know that $\sigma(s_2)\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]$, so by strong normalization (Lemma 4.12), the predicate in the active check goes to blame or to a value. If it goes to blame, we are done. If it goes to a value, that value must be true or false. If it goes to false, then the whole term goes to blame and we are done. If it goes to true, then the check will step to k. But $\sigma(s_2)\{x := k\} \longrightarrow_h^* \mathsf{true}$, so $k \in [\![\sigma(\{x{:}B \mid s_2\})]\!]$ by definition. Expansion (Lemma 4.9) completes the proof.

- $S_2 = x{:}S_{21} \to S_{22}$, so $S_1 = x{:}S_{11} \to S_{12}$. Let $q \in [\![\sigma(S_1)]\!]$; if it is blame we're done by Lemma 4.10, so let it be a value w. Let $q' \in [\![\sigma(S_{21})]\!]$; if it is blame we're done, so let it be a value w'. By F_CDecomp:

$$\langle \sigma(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow \sigma(S_{22})\{x := w'\}\rangle^l\, (w\, (\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l\, w'))$$

  By the IH, $\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l$ is semantically well-typed, so $\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l\, w' \in [\![\sigma(S_{11})]\!]$. By strong normalization (Lemma 4.12), this term reduces (and therefore parallel reduces, by Lemma `A4`) to some q''.

  We know that $w\, q'' \in [\![\sigma(S_{12})\{x := q''\}]\!]$ by assumption. $[\![\sigma(S_{12})\{x := q''\}]\!] = [\![\sigma(S_{12})\{x := \langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l\, w'\}]\!]$ by Corollary 4.18.

  Before applying the IH, we note that $\Delta \models S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\}$ and $\Delta \models S_{22}\{x := w'\}$ by Lemma 4.19. Then, by the IH we see that $\langle \sigma(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow \sigma(S_{22})\{x := w'\}\rangle^l$ is semantically well-kinded, so

$$\langle \sigma(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow \sigma(S_{22})\{x := w'\}\rangle^l\, (w\, w'') \in [\![\sigma(S_{22})\{x := w'\}]\!]$$

  and we are done. □

4.21 Lemma [Semantic type soundness]: If $\Delta \vdash s : S$ and $\Delta \vdash S$ then $\Delta \models s : S$ and $\Delta \models S$.

Proof: By induction on the typing and well-formedness derivations, using Corollary 4.18 in the S_App case and Lemma 4.14 in the S_Sub case.

<u>S_Var</u>: $\Delta \vdash x : S$ means that $x{:}S \in \Delta$; since $\Delta \models \sigma$, we know that $\sigma(x) \in [\![\sigma(\Delta(x))]\!]$, but $\sigma(\Delta(x)) = \sigma(S)$, so we are done.

<u>S_Const</u>: $\Delta \vdash k : \mathrm{ty_h}(k)$. We have $\Delta \models k : \mathrm{ty_h}(k)$ by the assumption that constants inhabit their types.

<u>S_Lam</u>: $\Delta \vdash \lambda x{:}S_1.\ s_2 : (x{:}S_1 \to S_2)$ and $\Delta, x{:}S_1 \vdash s_2 : S_2$. We must show that for all q, if $q \in [\![\sigma(S_1)]\!]$ then $\sigma(\lambda x{:}S_1.\ s_2)\ q \in [\![\sigma(S_2)\{x := q\}]\!]$.

By the IH, $\Delta, x{:}S_1 \models s_2 : S_2$. In particular, for all $w \in [\![S_1]\!]$, $\sigma(s_2)\{x := w\} \in [\![\sigma(S_2)\{x := w\}]\!]$.

Suppose $q \in [\![\sigma(S_1)]\!]$. We must show that $\sigma(s)\ q \in [\![\sigma(S_2)\{x := q\}]\!]$. If $q = \Uparrow l$, we're done by expansion (lemma 4.9). So suppose $q = w$. But $(\lambda x{:}\sigma(S_1).\ \sigma(s_2))\ w \longrightarrow_h^* \sigma(s_2)\{x := w\}$, which we know is in $[\![\sigma(S_2)\{x := w\}]\!]$.

<u>S_App</u>: $\Delta \vdash s_1\, s_2 : S_2$, and, by inversion, $\Delta \vdash s_1 : (x{:}S_1 \to S_2)$ and $\Delta \vdash s_2 : S_1$. By the IH, $\Delta \models s_1 : (x{:}S_1 \to S_2)$ and $\Delta \models s_2 : S_1$. We must show that, given $\Delta \models \sigma$, $\sigma(s_1\, s_2) \in [\![\sigma(S_2)\{x := s_2\}]\!]$. By the second IH, we know that $\sigma(s_2) \in [\![\sigma(S_1)]\!]$. By strong normalization (Lemma 4.12), $\sigma(s_2) \longrightarrow_h^* q$. By the first IH, we know that $\sigma(s_1)\ q \in [\![\sigma(S_2)\{x := q\}]\!]$, so we can conclude by expansion (Lemma 4.9) that $\sigma(s_1)\, \sigma(s_2) \in [\![\sigma(S_2)\{x := q\}]\!]$.

By Lemma `A6_2`, $\sigma(S_2)\{x := \sigma(s_2)\} \Rrightarrow^* \sigma(S_2)\{x := q\}$, since $s_2 \longrightarrow_h^* q$. By Corollary 4.18, $[\![\sigma(S_2)\{x := \sigma(s_2)\}]\!] = [\![\sigma(S_2)\{x := q\}]\!]$, so we find that $\sigma(s_1\, s_2) \in [\![\sigma(S_2)\{x := \sigma(s_2)\}]\!]$, and we are done.

<u>S_Blame</u>: $\Delta \vdash \Uparrow l : S$. By Lemma 4.10.

<u>S_Cast</u>: $\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2$. By the IH, $\Delta \models S_1$ and $\Delta \models S_2$; by inversion, $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$. Lemma 4.20 completes this case.

<u>S_Checking</u>: $\Delta \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}$. We must show that $\sigma(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) \in [\![\sigma(\{x{:}B \mid s_1\})]\!]$.

By the IH, we know that $\sigma(s_2) \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]$, so by strong normalization, we find that $\sigma(s_2) \longrightarrow_h^* q$ for some q_2. If $q_2 = \Uparrow l'$, then the whole term steps to $\Uparrow l'$. If $q_2 = w_2$, we know that w_2 is either true or false. In the latter case, the whole term steps to $\Uparrow l$, and we are done (by Lemma 4.10). If $w_2 = \mathsf{true}$, then the whole term steps to k. We see that the entire check steps to $k \in \mathcal{K}_B$. By inversion of the typing judgment, we know that $\Delta \vdash s_2 \supset s_1\{x := k\}$; since $\sigma(s_2) \longrightarrow_h^* \mathsf{true}$, it must be the case that $\sigma(s_1\{x := k\}) \longrightarrow_h^* \mathsf{true}$, so we see that $\sigma(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) \in [\![\sigma(\{x{:}B \mid s_1\})]\!]$.

<u>S_Sub</u>: $\Delta \vdash s : S_2$. By inversion, we know that $\Delta \vdash s : S_1$ and $\Delta \vdash S_1 <: S_2$. Our IH is that $\Delta \models s : S_1$. By soundness of subtyping (Lemma 4.14), we know that $\Delta \models S_1 <: S_2$. Unfolding these definitions (for our $\Delta \models \sigma$), we find that $\sigma(s) \in [\![\sigma(S_1)]\!]$ and $[\![\sigma(S_1)]\!] \subseteq [\![\sigma(S_2)]\!]$. This shows that $\sigma(s) \in [\![\sigma(S_2)]\!]$, and we are done.

<u>WF_Raw</u>: Immediate, since $\mathsf{true}\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]$ for all k.

<u>WF_Refine</u>: $\Delta \vdash \{x{:}B \mid s\}$; by inversion, $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$. By the IH, $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \models s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$.

We must show $\Delta \models \{x{:}B \mid s\}$. Let $\Delta \models \sigma$; we must show that, for all $k \in \mathcal{K}_B$, $\sigma(s)\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]$. Let $k \in \mathcal{K}_B$. We have $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \models \sigma\{x := k\}$, so $\sigma(s)\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]$, as desired.

<u>WF_Fun</u>: $\Delta \vdash x{:}S_1 \to S_2$. Let $\Delta \models \sigma$; we will show $\sigma(x{:}S_1 \to S_2) \in [\![\star]\!]$. First, $\sigma(S_1) \in [\![\star]\!]$ by the IH. Again by the IH, $\Delta, x{:}S_1 \vdash S_2$, so $\sigma(S_2)\{x := q\} \in [\![\star]\!]$ for any $q \in [\![\sigma(S_1)]\!]$. □

Once the semantic proof of type soundness is done, we can lift it to a standard proof of progress and preservation, using semantic soundness to handle the S_Imp case of the substitution lemma. However, the argument is still not completely straightforward: a similar problem arises in the proof of preservation, again in the application case. Consider a well-typed application $\Delta \vdash s_1\, s_2 : S_2\{x := s_2\}$. Suppose $s_2 \longrightarrow_h s_2'$. It is easy to reconstruct $\Delta \vdash s_1\, s_2 \;:\; S_2\{x := s_2'\}$, but that is not what we must show for preservation. How do $S_2\{x := s_2\}$ and $S_2\{x := s_2'\}$ relate? The semantic result of Corollary 4.18 is not enough—preservation requires us to be able to generate the typing derivation in the *syntactic* type system. To this end, we establish another property of parallel reduction: types that parallel-reduce are subtypes both ways round.

We proceed with a few more technical lemmas connecting parallel reduction and some of the judgments in our type system.

4.22 Lemma [Parallel reduction implies mutual implication]: If $s_1 \Rrightarrow s_2$, then $\Delta \vdash s_1 \supset s_2$ and $\Delta \vdash s_2 \supset s_1$.

Proof: Direct. Note that, by induction on Δ and P_Refl, if $\Delta \models \sigma$ then $\sigma(s_1) \Rrightarrow \sigma(s_2)$ by repeated application of Lemma A3. So $\sigma(s_1) \longrightarrow_h^* \texttt{true}$ iff $\sigma(s_2) \longrightarrow_h^* \texttt{true}$ by Lemma A20. $\square$

4.23 Lemma [Subtype narrowing for implication]: If $\Delta_1, x{:}S, \Delta_2 \vdash s_1 \supset s_2$ and $\Delta_1 \vdash S' <: S$ then $\Delta_1, x{:}S', \Delta_2 \vdash s_1 \supset s_2$.

Proof: Direct. By semantic type soundness (Lemma 4.21), we know that $\Delta_1 \models S' <: S$, i.e., if $\Delta_1 \models \sigma$, then $[\![\sigma_1(S')]\!] \subseteq [\![\sigma_1(S)]\!]$. So if $\Delta_1, x{:}S', \Delta_2 \models \sigma$, then we also have $\Delta_1, x{:}S, \Delta_2 \models \sigma$. By the implication assumption, we know that $\sigma(s_1) \longrightarrow_h^* \texttt{true}$ implies $\sigma(s_2) \longrightarrow_h^* \texttt{true}$. We conclude, then, that $\Delta_1, x{:}S', \Delta_2 \vdash s_1 \supset s_2$. $\square$

4.24 Lemma [Subtype narrowing]: If $\Delta_1, x{:}S, \Delta_2 \vdash S_1 <: S_2$ and $\Delta_1 \vdash S' <: S$ then $\Delta_1, x{:}S', \Delta_2 \vdash S_1 <: S_2$.

Proof: By induction on $\Delta_1, x{:}S, \Delta_2 \vdash S_1 <: S_2$.

Sub_Refine: By narrowing for the implication judgment (Lemma 4.23).

Sub_Fun: $S_1 = y{:}S_{11} \to S_{12}$ and $S_2 = y{:}S_{21} \to S_{22}$. By the IH, $\Delta_1, x{:}S', \Delta_2 \vdash S_{21} <: S_{11}$ and $\Delta_1, x{:}S', \Delta_2, y{:}S_{21} \vdash S_{12} <: S_{22}$, so by Sub_Fun we can conclude that $\Delta_1, x{:}S', \Delta_2 \vdash S_1 <: S_2$. $\square$

4.25 Lemma [Reflexivity of subtyping]: If $\Delta \vdash S$ then $\Delta \vdash S <: S$.

Proof: By induction on $|S|$.

- $S = \{x{:}B \mid s\}$. It is immediate that $\Delta \vdash s \supset s$.
- $S = x{:}S_1 \to S_2$. By the IHs. $\square$

4.26 Lemma: If $\Delta \vdash S_1$ and $S_1 \Rrightarrow S_2$ then $\Delta \vdash S_1 <: S_2$ and $\Delta \vdash S_2 <: S_1$.

Proof: By induction on $|S_1|$ (which is equal to $|S_2|$). The proof proceeds by case analysis on the parallel reduction rule used.

If $S_1 \Rrightarrow S_2$ by P_SRefl (i.e.,

P_SRefl: $S_1 = S_2$, so we have subtyping by Lemma 4.25.

P_SRefine: $S_1 = \{x{:}B \mid s_1\} \Rrightarrow \{x{:}B \mid s_2\} = S_2$. Parallel reduction implies mutual implication (Lemma 4.22), so $\Delta, x{:}\{x{:}B \mid \texttt{true}\} \vdash s_1 \supset s_2$ and vice versa; we finish by Sub_Refine.

P_SFun: $S_1 = x{:}S_{11} \to S_{12} \Rrightarrow x{:}S_{21} \to S_{22} = S_2$. By inversion, $S_{11} \Rrightarrow S_{21}$ and $S_{12} \Rrightarrow S_{22}$. Further, we know $\Delta \vdash S_{11}$ and $\Delta, x{:}S_{11} \vdash S_{12}$. By the IH, we know that:

$$\begin{array}{ll} \Delta \vdash S_{11} <: S_{21} & \Delta \vdash S_{21} <: S_{11} \\ \Delta, x{:}S_{11} \vdash S_{12} <: S_{22} & \Delta, x{:}S_{11} \vdash S_{22} <: S_{12} \end{array}$$

It is easy to construct $\Delta \vdash x{:}S_{21} \to S_{22} <: x{:}S_{11} \to S_{12}$ by Sub_Fun. Noticing that $\Delta \vdash S_{21} <: S_{11}$, we can see that $\Delta, x{:}S_{21} \vdash S_{12} <: S_{22}$ by narrowing (Lemma 4.24), and so $\Delta \vdash x{:}S_{11} \to S_{12} <: x{:}S_{21} \to S_{22}$. $\square$

We also prove that constants of base type inhabit the corresponding raw type.

4.27 Lemma [Trivial refinements of constants]: If $k \in \mathcal{K}_B$, then $k \in [\![\{x{:}B \mid \mathsf{true}\}]\!]$.

Proof: By definition: F_REFL allows us to derive $k \longrightarrow_h^* k$ and $\mathsf{true}\{x := k\} \longrightarrow_h^* \mathsf{true}$. □

The substitution lemma, necessary for preservation, has one complication: the operational judgment S_IMP requires the semantic type soundness theorem to show that a syntactically well-typed term can be used in a closing substitution. It is otherwise straightforward.

4.28 Lemma [Substitution (implication)]: If $\Delta_1, x{:}S, \Delta_2 \vdash s_1 \supset s_2$ and $\Delta_1 \vdash s : S$, then $\Delta_1, \Delta_2\{x := s\} \vdash s_1\{x := s\} \supset s_2\{x := s\}$.

Proof: Direct. For all $\Delta_1, x{:}S, \Delta_2 \models \sigma$, we have $\sigma(s_1) \longrightarrow_h^* \mathsf{true}$ implies $\sigma(s_2) \longrightarrow_h^* \mathsf{true}$.

Let $\Delta_1, \Delta_2\{x := s\} \models \sigma$. Suppose $\sigma(s_1\{x := s\}) \longrightarrow_h^* \mathsf{true}$. We must show $\sigma(s_2\{x := s\}) \longrightarrow_h^* \mathsf{true}$.

We can break σ apart into $\Delta_1 \models \sigma_1$ and $\sigma_1(\Delta_2)\{x := \sigma_1(s)\} \models \sigma_2$. Since $\Delta_1 \vdash s : S$, we know that $\sigma_1(s) \in [\![\sigma_1(S)]\!]$. Therefore $\Delta_1, x{:}\sigma_1(S) \models \sigma_1\{x := \sigma_1(s)\}$. Noting that $\sigma_1(\Delta_2\{x := s\}) = \sigma_1(\Delta_2)\{x := \sigma_1(s)\}$, we see $\Delta_1, x{:}S, \Delta_2 \models \sigma_1\{x := \sigma_1(s)\}\sigma_2$.

We know

$$\sigma_2(\sigma_1(s_1)\{x := \sigma_1(s)\}) = \sigma_2(\sigma_1(s_1\{x := s\})) = \sigma(s_1\{x := s\}) \longrightarrow_h^* \mathsf{true}$$

so by our original assumption,

$$\sigma_2(\sigma_1(s_2)\{x := \sigma_1(s)\}) = \sigma_2(\sigma_1(s_2\{x := s\})) = \sigma(s_2\{x := s\}) \longrightarrow_h^* \mathsf{true}$$

as well. □

4.29 Lemma [Substitution (subtyping)]: If $\Delta_1, x{:}S, \Delta_2 \vdash S_1 <: S_2$ and $\Delta_1 \vdash s : S$, then $\Delta_1, \Delta_2\{x := s\} \vdash S_1\{x := s\} <: S_2\{x := s\}$.

Proof: By induction on the subtyping derivation.

<u>SUB_REFINE</u>: $\Delta_1, x{:}S, \Delta_2 \vdash \{y{:}B \mid s_1\} <: \{y{:}B \mid s_2\}$. By substitution for implications (Lemma 4.28).

<u>SUB_ARROW</u>: By the IH. □

4.30 Lemma [Weakening]: If

- $\Delta \vdash s : S$,
- $\Delta \vdash S$, and
- $\Delta \cap \Delta' = \emptyset$

then $\Delta, \Delta' \vdash s : S$ and $\Delta, \Delta' \vdash S$.

Proof: By straightforward induction on s and $|S|$. □

4.31 Lemma [Substitution (typing and well-formedness)]: If $\Delta_1 \vdash s : S$ and

1. $\Delta_1, x{:}S, \Delta_2 \vdash s_1 : S_1$
2. $\Delta_1, x{:}S, \Delta_2 \vdash S_1$

then

1. $\Delta_1, \Delta_2\{x := s\} \vdash s_1\{x := s\} : S_1\{x := s\}$
2. $\Delta_1, \Delta_2\{x := s\} \vdash S_1\{x := s\}$

Proof: By mutual induction on the typing derivations.

<u>S_Var</u>: $\Delta_1, x{:}S, \Delta_2 \vdash y : S_1$. If $x = y$, then by weakening (Lemma 4.30). If not, then either $y{:}S_1 \in \Delta_1$ or $y{:}S_1 \in \Delta_2$. If the former is the case, then x isn't free in $S_1 = S_1\{x := s\}$; if the latter is the case, then $y{:}S_1\{x := s\} \in \Delta_2\{x := s\}$.

<u>S_Const</u>: Trivial.

<u>S_Lam</u>: $\Delta_1, x{:}S, \Delta_2 \vdash \lambda y{:}S_1.\ s_2 : S_1 \to S_2$. By the IH with $\Delta_2\{x := s\}, y{:}S_1\{x := s\}$, we have $\Delta_1, \Delta_2\{x := s\}, y{:}S_1\{x := s\} \vdash s_2\{x := s\} : S_2\{x := s\}$. By S_Lam, $\Delta_1, \Delta_2\{x := s\} \vdash (\lambda y{:}S_1.\ s_2)\{x := s\} : (y{:}S_1 \to S_2)\{x := s\}$.

<u>S_App</u>: By the IHs.

<u>S_Blame</u>: Trivial.

<u>S_Cast</u>: By the IH for well-formedness of S_1 and S_2.

<u>S_Checking</u>: Contradictory—only applies in the empty context.

<u>S_Sub</u>: By the IH and substitution for subtyping (Lemma 4.29).

<u>WF_Raw</u>: Trivial.

<u>WF_Refine</u>: $\Delta_1, x{:}S, \Delta_2 \vdash \{y{:}B \mid s_1\}$. By the IH with $\Delta_2\{x := s\}, y{:}\{y{:}B \mid \mathsf{true}\}$.

<u>WF_Fun</u>: By the IH. $\square$

4.32 Theorem [Preservation]: If $\emptyset \vdash s : S$ and $s \longrightarrow_h s'$ then $\emptyset \vdash s' : S$.

Proof: By induction on the typing derivation.

<u>S_App</u>: $\emptyset \vdash s_1\, s_2 : S_2\{x := s_2\}$. The proof proceeds by inversion on the step relation, treating the F_Reduce sub-cases as top-level cases:

> <u>F_Compat</u>: This is straightforward when $F = [\,]\ s_2$. But when $F = w_1\ [\,]$ and $s_2 \longrightarrow_h s_2'$, T_App can only produce the typing $\emptyset \vdash w_1\, s_2' : S_2\{x := s_2'\}$, where we need to see $S_2\{x := s_2\}$. But note that $s_2 \longrightarrow_h s_2'$, so we know that $S_2\{x := s_2\} \Rrightarrow S_2\{x := s_2'\}$ by Lemma `A5_2`. We then know by Lemma 4.26 that $\emptyset \vdash S_2\{x := s_2'\} <: S_2\{x := s_2\}$, in particular. So by T_Sub, we can see that $\emptyset \vdash w_1\, s_2' : S_2\{x := s_2\}$.
>
> <u>F_Blame</u>: By S_Blame.
>
> <u>F_Const</u>: Since constants are first-order, $\emptyset \vdash k : x{:}\{x{:}B_1 \mid s_1\} \to y{:}\{y{:}B_2 \mid s_2\} \to \ldots \to \{z{:}B_n \mid s_n\}$. By inversion, $\emptyset \vdash w : \{x{:}B_1 \mid s_1\}$, so $w = k' \in \mathcal{K}_{B_1}$. Since we assume that type assignments and denotations agree, $[\![k]\!](k')$ is defined and well-typed at $B_2 \to \ldots \to B_n$.
>
> <u>F_Beta</u>: By substitution (Lemma 4.31).
>
> <u>F_CCheck</u>: By S_Checking.
>
> <u>F_CDecomp</u>: By S_App, S_Cast, and the assumed typing.

<u>S_Checking</u>: $\emptyset \vdash \langle\{x{:}B \mid s_1\}, s_2, k\rangle^l : \{x{:}B \mid s_1\}$. By inversion on the step relation, treating the F_Reduce sub-cases as top-level cases:

> <u>F_Compat</u>: $s_2 \longrightarrow_h s_2'$. We get everything from our assumptions and the IH except for $\emptyset \vdash s_2' \supset s\{x := k\}$. But if $s_2 \longrightarrow_h^* \mathsf{true}$ and $s_2 \longrightarrow_h s_2'$, then $s_2' \longrightarrow_h^* \mathsf{true}$ by coevaluation (Coevaluation 4.8), so we have that as well.
>
> <u>F_Blame or F_Fail</u>: By S_Blame.
>
> <u>F_OK</u>: We know by inversion that $\emptyset \vdash \mathsf{true} \supset s_1\{x := k\}$, so by the fact that constants have most-specific types, S_Sub and the assumption that constants have closed, well-formed types, and S_Const we can find $\emptyset \vdash k : \{x{:}B \mid s_1\}$.

S_SUB: By the IH. $\square$

The proof of progress, fortunately, is entirely straightforward.

4.33 Theorem [Progress]: If $\emptyset \vdash s : S$ then either $s \longrightarrow_h s'$ or $s = q$, i.e., $s = \Uparrow l$ or $s = w$.

Proof: By induction on the typing derivation. S_VAR is contradictory, as there is no such typing. S_CONST, S_LAM, and S_CAST are values. S_BLAME is blame.

S_APP: $\emptyset \vdash s_1\, s_2 : S_2\{x := s_2\}$ and $\emptyset \vdash s_1 : (x{:}S_1 \to S_2)$ and $\emptyset \vdash s_2 : S_1$. Either $s_1 \longrightarrow_h s_1'$ or $s_1 = r_1$. The first case takes a step by F_COMPAT. If $s_1 = \Uparrow l$, then we take a step by F_BLAME.

So $s_1 = w_1$. Either $s_2 \longrightarrow_h s_2'$ or $s_2 = r_2$. The first case takes a step by F_COMPAT. If $s_2 = \Uparrow l$, we take a step by F_BLAME. So $s_2 = w_2$. To see how s steps, we perform case analysis on w_1:

- $w_1 = k$. By inversion, $\emptyset \vdash w : \{x{:}B_1 \mid s_1\}$, so $w = k' \in \mathcal{K}_{B_1}$. Since we assume that type assignments and denotations agree, $[\![k]\!](k')$ is defined. So s steps by F_REDUCE and F_CONST.
- $w_1 = \lambda x{:}S_1.\ s_{12}$. By F_BETA.
- $w_1 = \langle\{x{:}B \mid s_{11}\} \Rightarrow \{x{:}B \mid s_{12}\}\rangle^l$. By inversion of the typing, we know that $w_2 = k$, and so s takes a step by F_REDUCE and F_CCHECK.
- $w_1 = \langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l$. $w_1\, w_2$ is a value.
- $w_1 = \langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\, w_1'$. s takes a step by F_REDUCE and F_CDECOMP.

S_CHECKING: $\emptyset \vdash \langle\{x{:}B \mid s_1\}, s_2, k\rangle^l : \{x{:}B \mid s_1\}$. By the IH, either $s_2 \longrightarrow_h s_2'$ or $s_2 = q_2$. In the former case, we step by F_COMPAT. If $s_2 = \Uparrow l'$, then we step by F_BLAME. If $s_2 = w_2$, then $\emptyset \vdash w_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ means w_2 is either true or false, in which case it steps by F_REDUCE and F_OK or F_FAIL, respectively.

S_SUB: By the IH. $\square$

$\boxed{\vdash \Delta}$

$$\frac{}{\vdash \emptyset}\ \textsc{S\_Empty} \qquad \frac{\vdash \Delta \qquad \Delta \vdash S}{\vdash \Delta, x{:}S}\ \textsc{S\_ExtVar}$$

$\boxed{\Delta \vdash s : S}$

$$\frac{x{:}S \in \Delta}{\Delta \vdash x : S}\ \textsc{S\_Var} \qquad \frac{}{\Delta \vdash k : \mathrm{ty_h}(k)}\ \textsc{S\_Const}$$

$$\frac{\Delta \vdash S_1 \qquad \Delta, x{:}S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x{:}S_1.\ s_2 : (x{:}S_1 \to S_2)}\ \textsc{S\_Lam} \qquad \frac{\Delta \vdash s_1 : (x{:}S_1 \to S_2) \qquad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\, s_2 : S_2\{x := s_2\}}\ \textsc{S\_App}$$

$$\frac{\begin{array}{c}\Delta \vdash S_1 \qquad \Delta \vdash S_2 \\ \lfloor S_1 \rfloor = \lfloor S_2 \rfloor\end{array}}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2}\ \textsc{S\_Cast} \qquad \frac{\begin{array}{c}\Delta \vdash s : S_1 \qquad \Delta \vdash S_2 \\ \Delta \vdash S_1 <: S_2\end{array}}{\Delta \vdash s : S_2}\ \textsc{S\_Sub}$$

$$\frac{\begin{array}{c}\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\} \qquad \emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \\ \emptyset \vdash \{x{:}B \mid s_1\} \qquad \emptyset \vdash s_2 \supset s_1\{x := k\}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}}\ \textsc{S\_Checking}$$

$\boxed{\Delta \vdash s_1 \supset s_2}$

$$\frac{\forall \sigma.\ ((\Delta \models \sigma \,\mathrm{and}\, \sigma(s_1) \longrightarrow_h^* \mathsf{true}) \quad \text{implies} \quad \sigma(s_2) \longrightarrow_h^* \mathsf{true})}{\Delta \vdash s_1 \supset s_2}\ \textsc{S\_Imp}$$

$\boxed{\Delta \vdash S}$

$$\frac{}{\Delta \vdash \{x{:}B \mid \mathsf{true}\}}\ \textsc{WF\_Raw} \qquad \frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\Delta \vdash \{x{:}B \mid s\}}\ \textsc{WF\_Refine}$$

$$\frac{\Delta \vdash S_1 \qquad \Delta, x{:}S_1 \vdash S_2}{\Delta \vdash x{:}S_1 \to S_2}\ \textsc{WF\_Fun}$$

$\boxed{\Delta \vdash S_1 <: S_2}$

$$\frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 \supset s_2}{\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}}\ \textsc{Sub\_Refine} \qquad \frac{\Delta \vdash S_{21} <: S_{11} \qquad \Delta, x{:}S_{21} \vdash S_{12} <: S_{22}}{\Delta \vdash x{:}S_{11} \to S_{12} <: x{:}S_{21} \to S_{22}}\ \textsc{Sub\_Fun}$$

$\boxed{\Delta \models \sigma}$

$$\frac{}{\emptyset \models \emptyset}\ \textsc{CS\_Empty} \qquad \frac{s \in [\![S]\!] \qquad \Delta\{x := s\} \models \sigma}{x{:}S, \Delta \models \sigma\{x := s\}}\ \textsc{CS\_Ext}$$

Figure 18: Typing rules for dependent λ_{H}

$\boxed{s_1 \Rrightarrow s_2}$ s_1 parallel-reduces to s_2

$$\frac{}{s \Rrightarrow s} \quad \text{P\_Refl}$$

$$\frac{w \Rrightarrow w'}{k\ w \Rrightarrow [\![k]\!](w')} \quad \text{P\_RConst}$$

$$\frac{s_{12} \Rrightarrow s'_{12} \qquad w_2 \Rrightarrow w'_2}{(\lambda x{:}S.\ s_{12})\ w_2 \Rrightarrow s'_{12}\{x := w'_2\}} \quad \text{P\_RBeta}$$

$$\frac{s_2 \Rrightarrow s'_2}{\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\ k \Rrightarrow \langle \{x{:}B \mid s'_2\}, s'_2\{x := k\}, k\rangle^l} \quad \text{P\_RCCheck}$$

$$\frac{}{\langle \{x{:}B \mid s\}, \mathsf{true}, k\rangle^l \Rrightarrow k} \quad \text{P\_ROK}$$

$$\frac{}{\langle \{x{:}B \mid s\}, \mathsf{false}, k\rangle^l \Rrightarrow \Uparrow l} \quad \text{P\_RFail}$$

$$\frac{\begin{array}{cc} S_{11} \Rrightarrow S'_{11} & S_{12} \Rrightarrow S'_{12} \\ S_{21} \Rrightarrow S'_{21} & S_{22} \Rrightarrow S'_{22} \\ w_1 \Rrightarrow w'_1 & w_2 \Rrightarrow w'_2 \end{array}}{(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\ w_1)\ w_2 \Rrightarrow \langle S'_{12}\{x := \langle S'_{21} \Rightarrow S'_{11}\rangle^l\ w'_2\} \Rightarrow S'_{22}\{x := w'_2\}\rangle^l\ (w'_1\ (\langle S'_{21} \Rightarrow S'_{11}\rangle^l\ w'_2))} \quad \text{P\_RCDecomp}$$

$$\frac{S_1 \Rrightarrow S'_1 \qquad s_{12} \Rrightarrow s'_{12}}{\lambda x{:}S_1.\ s_{12} \Rrightarrow \lambda x{:}S'_1.\ s'_{12}} \quad \text{P\_Lam}$$

$$\frac{s_1 \Rrightarrow s'_1 \qquad s_2 \Rrightarrow s'_2}{s_1\ s_2 \Rrightarrow s'_1\ s'_2} \quad \text{P\_App}$$

$$\frac{S_1 \Rrightarrow S'_1 \qquad S_2 \Rrightarrow S'_2}{\langle S_1 \Rightarrow S_2\rangle^l \Rrightarrow \langle S'_1 \Rightarrow S'_2\rangle^l} \quad \text{P\_Cast}$$

$$\frac{S \Rrightarrow S' \qquad s \Rrightarrow s'}{\langle S, s, k\rangle^l \Rrightarrow \langle S', s', k\rangle^l} \quad \text{P\_Check}$$

$$\frac{}{F\,[\Uparrow l] \Rrightarrow \Uparrow l} \quad \text{P\_Blame}$$

$\boxed{S_1 \Rrightarrow S_2}$ S_1 parallel-reduces to S_2

$$\frac{}{S \Rrightarrow S} \quad \text{P\_SRefl}$$

$$\frac{s \Rrightarrow s'}{\{x{:}B \mid s\} \Rrightarrow \{x{:}B \mid s'\}} \quad \text{P\_SRefine}$$

$$\frac{S_1 \Rrightarrow S'_1 \qquad S_2 \Rrightarrow S'_2}{x{:}S_1 \to S_2 \Rrightarrow x{:}S'_1 \to S'_2} \quad \text{P\_SFun}$$

Figure 19: Parallel reduction for dependent λ_H

Term translation

$$\begin{array}{rcl}
\psi(x) & = & x \\
\psi(k) & = & k \\
\psi(\lambda x{:}S.\ s) & = & \lambda x{:}\lfloor S \rfloor.\ \psi(s) \\
\psi(s_1\, s_2) & = & \psi(s_1)\, \psi(s_2) \\
\psi(\Uparrow l) & = & \Uparrow l \\
\psi(\langle S_1 \Rightarrow S_2 \rangle^l) & = & \langle \psi^l(S_1, S_2) \rangle^{l,l} \\
\psi(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) & = & \langle \{x{:}B \mid \psi(s_1)\}, \psi(s_2), k \rangle^l
\end{array}$$

Type-to-contract translation

$$\begin{array}{rcl}
\psi^l(\{x{:}B \mid s_1\}, \{x{:}B \mid s_2\}) & = & \{x{:}B \mid \psi(s_2)\} \\
\psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) & = & \\
\multicolumn{3}{r}{x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\, x\}, S_{22})}
\end{array}$$

Figure 20: ψ mapping dependent λ_{H} to dependent λ_{C}

5 Translating $\lambda_{\mathbf{H}}$ to $\lambda_{\mathbf{C}}$: dependent ψ

In this section, we formally define ψ for the dependent versions of λ_{C} and λ_{H}. We prove that ψ is type preserving and induces behavioral correspondence.

The full definition of ψ is in Figure 20. Most terms are translated homomorphically. In abstractions, the annotation is translated by erasing the refined λ_{H} type to its simple-type skeleton. As mentioned in Section 3, the trickiest part is the translation of casts between function types: we are careful to mimic the behavior of the F_CDecomp rule—when generating the codomain contract from a cast between two function types, we perform the same asymmetric substitution as F_CDecomp. Since ψ on casts inserts new casts, we need to pick a blame label to use: $\psi(\langle S_1 \Rightarrow S_2 \rangle^l)$ passes l as an index to $\psi^l(S_1, S_2)$ to use when generating casts.

We prove that s and $\psi(s)$ behaviorally correspond using logical relations. We then show that ψ is type preserving—this is necessary because the typing rules T_Checking and S_Checking refer to the operational semantics.

We present the *term correspondence* relation $t \sim s : T$ in Figure 21; it is a logical relation, defined as a fixpoint over its λ_{C}-type index. The *contract/cast correspondence* $c \sim S_1 \supset^l S_2 : T$ relates contracts and pairs of λ_{H} types. We define it using the term correspondence in the base type case, and following the pattern of F_CDecomp in the function case. Since it inserts a cast in the function cast, we index it with a label, just like ψ. Note that the correspondence is blame-exact, relating λ_{C} and λ_{H} terms that either blame the same label or go to corresponding values. We define closing substitutions ignoring the contracts in the context; these permit us to lift the relation to open terms in the standard way. The behavioral correspondence is quite strong: it is easy to check that, if $t \sim s : B$, then t and s either both go to $k \in \mathcal{K}_B$ or both go to $\Uparrow l$.

We begin with some standard properties of the term correspondence relation.

5.1 Lemma [Expansion and contraction]: If:

- $t \longrightarrow_c^* t'$, and
- $s \longrightarrow_h^* s'$

then $t \sim s : T$ iff $t' \sim s' : T$.

Proof: By coevaluation for $\longrightarrow_c^*$ (Corollary 4.3) and $\longrightarrow_h^*$ (Corollary 4.8). □

Result correspondence

$$\begin{array}{c} r \ \approx\ q \ :\ B \iff \\ r = q = \Uparrow l \quad \vee \\ r = q = k \in \mathcal{K}_B \end{array}$$

$$\begin{array}{c} r \ \approx\ q \ :\ T_1 \to T_2 \iff \\ r = q = \Uparrow l \quad \vee \\ (r = v \wedge q = v \quad \wedge \\ \forall t' \sim s' : T_1.\ r\, t' \ \sim\ q\, s' \ :\ T_2) \end{array}$$

Term correspondence

$$\begin{array}{c} t \ \sim\ s \ :\ T \iff \\ t \ \longrightarrow_c^* \ r \wedge s \ \longrightarrow_h^* \ q \wedge r \ \approx\ q \ :\ T \end{array}$$

Contract / type correspondence

$$\begin{array}{c} \{x{:}B \mid t\} \sim \{x{:}B \mid s_1\} \supset^l \{x{:}B \mid s_2\} : B \iff \\ \forall k \in \mathcal{K}_B.\ t\{x := k\} \ \sim\ s_2\{x := k\} \ :\ \mathsf{Bool} \end{array}$$

$$\begin{array}{c} x{:}c_1 \mapsto c_2 \sim x{:}S_{11} \to S_{12} \supset^l S_{21} \to S_{22} : T_1 \to T_2 \iff \\ c_1 \sim S_{21} \supset^l S_{11} : T_1 \quad \wedge \\ \forall t \ \sim\ s \ :\ T_1. \\ c_2\{x := t\} \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, s\} \supset^l S_{22}\{x := s\} : T_2 \end{array}$$

$\boxed{\Gamma \models \delta}$

$$\frac{}{\emptyset \models \emptyset} \qquad \text{R\_CEmpty}$$

$$\frac{\Gamma \models \delta \qquad t \ \sim\ s \ :\ T}{\Gamma, x{:}T \models \delta\{x := t, s\}} \qquad \text{R\_CSubstT}$$

$$\frac{\Gamma \models \delta \qquad t \ \sim\ s \ :\ \lfloor c \rfloor}{\Gamma, x{:}c \models \delta\{x := t, s\}} \qquad \text{R\_CSubstC}$$

Lifted to open terms

$$\begin{array}{rcl} \Gamma \vdash t \ \sim\ s \ :\ T & \iff & \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(t) \ \sim\ \delta_2(s) \ :\ T) \\ \Gamma \vdash c \sim S_1 \supset^l S_2 : T & \iff & \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(c) \sim \delta_2(S_1) \supset^l \delta_2(S_2) : T) \end{array}$$

Figure 21: Blame-exact correspondence

5.2 Lemma [Blame corresponds to blame]: For all T, $\Uparrow l \sim \Uparrow l : T$.

Proof: By the definition of $\approx$ and reflexivity of the multi-step relations. $\square$

5.3 Lemma [Constants correspond to themselves]: For all k, $k \approx k : \mathrm{ty_c}(k)$.

Proof: By induction on $\mathrm{ty_c}(k)$.

$\underline{\mathrm{ty_c}(k) = B}$: By definition.

$\underline{\mathrm{ty_c}(k) = B_1 \to B_2 \to \ldots \to B_n}$: We wish to show that $k \approx k : B_1 \to B_2 \to \ldots \to B_n$. Let $t \sim s : B_1$. By definition $t \longrightarrow_c^* r$ and $s \longrightarrow_h^* q$ such that $r \approx q : B_1$. By expansion (Lemma 5.1), it suffices to show $k\,r \sim k\,q : B_2 \to \ldots \to B_n$.

If $r = q = \Uparrow l$, then $k \Uparrow l$ reduces to $\Uparrow l$ on both sides. By Lemma 5.2, $\Uparrow l \sim \Uparrow l : B_2 \to \ldots \to B_n$; the applications are related by expansion (Lemma 5.1).

Otherwise, we have $r = q = k' \in \mathcal{K}_{B_1}$ because constants are the only values related at B_1. Since denotations are total, $[\![k]\!](k')$ is defined. Moreover, $[\![k]\!](k')$ is either a constant k'' or $\Uparrow l$, by definition. In the latter case the applications are related because they go to blame. In the former, $\mathrm{ty_c}(k'') = B_2 \to \ldots \to B_n$, so $k'' \approx k'' : B_2 \to \ldots \to B_n$ by the IH. $\square$

As a corollary of Lemma 5.2 and Lemma 5.1, if two terms evaluate to blame, then they correspond. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

5.4 Lemma [Corresponding terms coevaluate]: If $t \sim s : T$ then $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ or $t \longrightarrow_c^* \Uparrow l$ and $s \longrightarrow_h^* \Uparrow l$; moreover, $t \longrightarrow_c^* r$ and $s \longrightarrow_h^* q$ such that $r \approx q : T$.

Proof: We find the evaluation to related results by definition. We can see by the definition of the relation on results that $r = \Uparrow l$ iff $q = \Uparrow l$, and likewise for values. $\square$

5.5 Lemma [Contract/Cast correspondence]: If

- $c \sim S_1 \supset^l S_2 : T$ and
- $t \sim s : T$

then $\langle c \rangle^{l,l}\, t \sim \langle S_1 \Rightarrow S_2 \rangle^l\, s : T$.

Proof: By induction on T. We reason via expansion (Lemma 5.1), showing that the initial terms reduce to corresponding terms.

- $T = B$, so $c = \{x{:}B \mid t_1\}$, $S_1 = \{x{:}B \mid s_1\}$, and $S_2 = \{x{:}B \mid s_2\}$. Since $t \sim s : B$, we know that they either both reduce to $k \in \mathcal{K}_B$ or $\Uparrow l'$. If the latter is the case, we are done. So suppose $t \longrightarrow_c^* k$ along with $s \longrightarrow_h^* k$.

  We can step our terms into active checks as follows, then:

$$\begin{array}{rcl} \langle \{x{:}B \mid t_1\} \rangle^{l,l}\, t & \longrightarrow_c^* & \langle \{x{:}B \mid t_1\}, t_1\{x := k\}, k \rangle^l \\ \langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} \rangle^l\, s & \longrightarrow_h^* & \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k \rangle^l \end{array}$$

  By inversion of the contract/cast correspondence, we know that $t_1\{x := k\} \sim s_2\{x := k\} : \mathsf{Bool}$, so these terms go to blame or to a Bool together. If they go to $\Uparrow l'$, we are done. If they go to false, then both the obligation and the cast will go to $\Uparrow l$. Finally, if they both go to true, then both terms will evaluate to k.

- $T = T_1 \to T_2$. $c = x{:}c_1 \mapsto c_2$, $S_1 = x{:}S_{11} \to S_{12}$, and $S_2 = x{:}S_{21} \to S_{22}$. We know by inversion of the contract/cast relation that $c_1 \sim S_{21} \supset^l S_{11} : T_1$ and that for all $t \sim s : T_1$, $c_2\{x := t\} \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, s\} \supset^l S_{22}\{x := s\} : T_2$. We want to prove that $\langle c\rangle^{l,l} \sim \langle S_1 \Rightarrow S_2\rangle^l\, s : T_1 \to T_2$. First, we can assume $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ where $v \sim w : T_1 \to T_2$—if not, both cast and contracted terms go to blame and we are done.

  We show that the decomposition of the contract and cast terms correspond for all inputs. Let $t' \sim s' : T_1$. Again, we can assume that they reduce to $v' \sim w' : T_1$, or else we are done by blame lifting. On the λ_{C} side, we have

$$(\langle c\rangle^{l,l}\, t)\, t' \longrightarrow_c^* \langle c_2\{x := v'\}\rangle^{l,l}\,(v\,(\langle c_1\rangle^{l,l}\, v'))$$

  In λ_{H}, we find

$$(\langle S_1 \Rightarrow S_2\rangle^l\, s)\, s' \longrightarrow_h^* \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w\} \Rightarrow S_{22}\{x := w'\}\rangle^l\,(w\,(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w'))$$

  By the IH, we know that $\langle c_1\rangle^{l,l}\, v' \sim \langle S_{21} \Rightarrow S_{11}\rangle^l\, w' : T_1$. Since $v \sim w : T_1 \to T_2$, we have $v\,(\langle c_1\rangle^{l,l}\, v') \sim w\,(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w') : T_2$. Again by the IH, we can see that $\langle c_2\{x := v'\}\rangle^{l,l}\,(v\,(\langle c_1\rangle^{l,l}\, v')) \sim \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow S_{22}\{x := w'\}\rangle^l\, w\,(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w') : T_2$. □

We prove three more lemmas of a technical nature.

5.6 Lemma [Skeletal equality of subtypes]: If $\Delta \vdash S_1 <: S_2$, then $\lfloor S_1\rfloor = \lfloor S_2\rfloor$.

Proof: By induction on the subtyping derivation.

SUB_REFINE: $\lfloor S_1\rfloor = \lfloor S_2\rfloor = B$.

SUB_FUN: By the IH, $\lfloor S_{21}\rfloor = \lfloor S_{11}\rfloor = T_1$ and $\lfloor S_{12}\rfloor = \lfloor S_{22}\rfloor = T_2$, so $\lfloor x{:}S_{11} \to S_{12}\rfloor = \lfloor x{:}S_{21} \to S_{22}\rfloor = T_1 \to T_2$. □

5.7 Lemma: If $\lfloor S_1\rfloor = \lfloor S_2\rfloor = T$, then $\lfloor \psi^l(S_1, S_2)\rfloor = T$.

Proof: By induction on the $\lfloor S_2\rfloor$. □

5.8 Lemma: If $\Delta_1 \vdash S_1$ and $\Delta_1 \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor$ then:

1. If $\Delta_1, x{:}S_1, \Delta_2 \vdash s : S$ then $\Delta_1, x{:}S_2, \Delta_2\{x := \langle S_2 \Rightarrow S_1\rangle^l\, x\} \vdash s\{x := \langle S_2 \Rightarrow S_1\rangle^l\, x\} : S\{x := \langle S_2 \Rightarrow S_1\rangle^l\, x\}$
2. If $\Delta_1, x{:}S_1, \Delta_2 \vdash S$ then $\Delta_1, x{:}S_2, \Delta_2\{x := \langle S_2 \Rightarrow S_1\rangle^l\, x\} \vdash S\{x := \langle S_2 \Rightarrow S_1\rangle^l\, x\}$

Proof: By straightforward mutual induction on the typing derivations. The initial assumptions give us $\Delta_1, x{:}S_2 \vdash \langle S_2 \Rightarrow S_1\rangle^l\, x : S_1$—the rest of the proof proceeds like substitution, Lemma 4.31. □

We can use these relations to show that s and its translation $\psi(s)$ correspond behaviorally—i.e., that ψ faithfully translates the λ_{H} semantics. We must choose the subject of induction carefully, however, to ensure that we can apply the IH in the case for function casts. An induction on the height of the well-formedness derivation is tricky because of the substitution that ψ does in the function case. Instead, we do induction on the depth of ψ's recursion, (and also derivation height, for the S_SUB case).

5.9 Theorem [Behavioral correspondence]:

1. If $\psi(s) = t$ and $\Delta \vdash s : S$ then $\lfloor \Delta\rfloor \vdash t \sim s : \lfloor S\rfloor$.
2. If $\psi^l(S_1, S_2) = c$ and $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor = \lfloor S\rfloor$, then $\lfloor \Delta\rfloor \vdash c \sim S_1 \supset^l S_2 : \lfloor S\rfloor$ (for all l).

Proof: By induction on the lexicographically ordered pairs (m, n), where m is the depth of the recursion of the translation $\psi(s)$ (for part 1) or $\psi^l(S_1, S_2)$ (for part 2) and n is either $|\Delta \vdash s : S|$ (for part 1) or $|\Delta \vdash S_1| + |\Delta \vdash S_2|$ (for part 2). The first component decreases in all uses of the IH except for the S_Sub case, where only the second component decreases.

Part (1) of the proof proceeds by case analysis on the final rule used in the typing derivation $\Delta \vdash s : S$. Which rule was used determines the shape of $\psi(s)$ in all cases but S_Sub.

S_Var: $\Delta \vdash x : S$ and $\psi(x) = x$. By inversion $x{:}S \in \Delta$, so $x{:}\lfloor S \rfloor \in \lfloor \Delta \rfloor$. Let $\lfloor \Delta \rfloor \models \delta$—then $\delta_1(x) \sim \delta_2(x) : \lfloor S \rfloor$ by assumption.

S_Const: $\Delta \vdash k : \mathrm{ty_h}(k)$ and $\psi(k) = k$. $k \sim k : B$ by Lemma 5.3.

S_Lam: $\Delta \vdash \lambda x{:}S_1.\ s_{12} : (x{:}S_1 \to S_2)$ and $\psi(\lambda x{:}S_1.\ s_{12}) = \lambda x{:}\lfloor S_1 \rfloor.\ \psi(s_{12})$. By inversion $\Delta, x{:}S_1 \vdash s_{12} : S_2$.

The IH shows $\lfloor \Delta \rfloor, x{:}\lfloor S_1 \rfloor \vdash \psi(s_{12}) \sim s_{12} : \lfloor S_2 \rfloor$.

Let $\lfloor \Delta \rfloor \models \delta$. We must show that $\delta_1(\lambda x{:}\lfloor S_1 \rfloor.\ \psi(s_{12})) \sim \delta_2(\lambda x{:}S_1.\ s_{12}) : \lfloor S_1 \to S_2 \rfloor$. Let $t \sim s : \lfloor S_1 \rfloor$; we want to prove $\delta_1(\lambda x{:}\lfloor S_1 \rfloor.\ \psi(s_{12}))\ t \sim \delta_2(\lambda x{:}S_1.\ s_{12})\ s : \lfloor S_2 \rfloor$. If t and s go to $\Uparrow l$, then so do both of the applications, and we're done by Lemma 5.2. So let them go to $v \sim w : \lfloor S_1 \rfloor$. Then we can step both of the terms:

$$\begin{aligned} \delta_1(\lambda x{:}\lfloor S_1 \rfloor.\ \psi(s_{12}))\ t &\longrightarrow_c^* \delta_1(\psi(s_{12}))\{x := v\} \\ \delta_2(\lambda x{:}S_1.\ s_{12})\ s &\longrightarrow_h^* \delta_2(s_{12})\{x := w\} \end{aligned}$$

We can instantiate the IH to see that these terms correspond. The applications evaluate to corresponding terms, so they correspond (Lemma 5.1). Since the applications correspond, so do the two lambdas.

S_App: $\Delta \vdash s_1\, s_2 : S_2\{x := s_2\}$ and $\psi(s_1\, s_2) = \psi(s_1)\, \psi(s_2)$. By inversion, $\Delta \vdash s_1 : (x{:}S_1 \to S_2)$ and $\Delta \vdash s_2 : S_1$.

By the IH, $\lfloor \Delta \rfloor \vdash \psi(s_1) \sim s_1 : \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor$ and $\lfloor \Delta \rfloor \vdash \psi(s_2) \sim s_2 : \lfloor S_1 \rfloor$.

Let $\lfloor \Delta \rfloor \models \delta$; we must show that $\delta_1(\psi(s_1\, s_2)) \sim \delta_2(s_1\, s_2) : \lfloor S_2 \rfloor$. By definition, $\delta_1(\psi(s_1))$ and $\delta_2(s_1)$ evaluate to $r_1 \approx q_1 : T_1 \to T_2$. If $r_1 = q_1 = \Uparrow l$, then both applications go to blame and we are done (Lemma 5.2). So $r_1 = v_1$ and $q_1 = w_1$ and $v_1\, \delta_1(\psi(s_2)) \sim w_1\, \delta_2(s_2) : T_2$. We can see by expansion (Lemma 5.1) that $\delta_1(\psi(s_1)\, \psi(s_2)) \sim \delta_2(s_1\, s_2) : T_2$.

S_Blame: $\Delta \vdash \Uparrow l : S$ and $\psi(\Uparrow l) = \Uparrow l$. By Lemma 5.2, $\lfloor \Delta \rfloor \vdash \Uparrow l \sim \Uparrow l : \lfloor S \rfloor$.

S_Cast: $\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2$ and $\psi(\langle S_1 \Rightarrow S_2 \rangle^l) = \langle \psi^l(S_1, S_2) \rangle^{l,l}$. By inversion, $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$.

By the IH for proposition (2), $\lfloor \Delta \rfloor \vdash \psi^l(S_1, S_2) \sim S_1 \supset^l S_2 : \lfloor S_2 \rfloor$.

Let $\lfloor \Delta \rfloor \models \delta$; we must show $\delta_1(\langle \psi^l(S_1, S_2) \rangle^{l,l}) \sim \delta_2(\langle S_1 \Rightarrow S_2 \rangle^l) : \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor$. Let $t \sim s : \lfloor S_1 \rfloor$. We have $\delta_1(\langle \psi^l(S_1, S_2) \rangle^{l,l})\ t \sim \delta_2(\langle S_1 \Rightarrow S_2 \rangle^l)\ s : \lfloor S_2 \rfloor$ by Lemma 5.5.

S_Checking: $\emptyset \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}$ and $\psi(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) = \langle \{x{:}B \mid \psi(s_1)\}, \psi(s_2), k \rangle^l$. By inversion we have $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\}$, so $k \in \mathcal{K}_B$.

By the IH, $\psi(s_2) \sim s_2 : \mathsf{Bool}$. These two terms coevaluate to blame or a boolean constant. There are three cases, all of which result in the active checks evaluating to $\approx$-corresponding values:

- If they go to $\Uparrow l'$, then the checks do too, and $\Uparrow l' \approx \Uparrow l' : B$.
- If they go to false, then the checks go to $\Uparrow l$, and $\Uparrow l \approx \Uparrow l : B$.
- If they go to true, then the checks go to $k \in \mathcal{K}_B$, and $k \approx k : B$.

S_SUB: $\Delta \vdash s : S$; we do not know anything about the shape of $\psi(s)$. By inversion, $\Delta \vdash s : S'$ and $\Delta \vdash S' <: S$. By Lemma 5.6, $\lfloor S' \rfloor = \lfloor S \rfloor$.

Since the sub-derivation $\Delta \vdash s : S'$ is smaller, by the IH $\lfloor \Delta \rfloor \vdash \psi(s) \sim s : \lfloor S' \rfloor$. But $\lfloor S' \rfloor = \lfloor S \rfloor$, so we're done.

Part (2) of this proof proceeds by cases on $\psi^l(S_1, S_2) = c$.

- $\psi^l(S_1, \{x{:}B \mid s_2\}) = \{x{:}B \mid \psi(s_2)\}$, noting $S_2 = \{x{:}B \mid s_2\}$. By inversion of $\Delta \vdash \{x{:}B \mid s_2\}$, we have $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$.

  By the IH for proposition (1), $\lfloor \Delta \rfloor, x{:}B \vdash \psi(s_2) \sim s_2 : \mathsf{Bool}$.

  We must show $\lfloor \Delta \rfloor \vdash \{x{:}B \mid \psi(s_2)\} \sim S_1 \supset^l \{x{:}B \mid s_2\} : B$. Let $\lfloor \Delta \rfloor \models \delta$; we prove that $\delta_1(\{x{:}B \mid \psi(s_2)\}) \sim \delta_2(S_1) \supset^l \delta_2(\{x{:}B \mid s_2\}) : B$, i.e., for all $k \in \mathcal{K}_B$, that $\delta_1(\psi(s_2))\{x := k\} \sim \delta_2(s_2)\{x := k\} : \mathsf{Bool}$. Since $k \sim k : B$, we can see this last by the IH.

- $\psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) = x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22})$. We can see $S_2 = x{:}S_{21} \to S_{22}$ and so $S_1 = x{:}S_{11} \to S_{12}$, where $\lfloor S_{21} \rfloor = \lfloor S_{11} \rfloor$ and $\lfloor S_{22} \rfloor = \lfloor S_{12} \rfloor$. By inversion, we have the following well-formedness derivations:

  $$\begin{array}{rr} \Delta \vdash S_{21} & \Delta \vdash S_{11} \\ \Delta, x{:}S_{21} \vdash S_{22} & \Delta, x{:}S_{22} \vdash S_{12} \end{array}$$

  We can apply the IH (contravariantly) to see

  $$\lfloor \Delta \rfloor \vdash \psi^l(S_{21}, S_{11}) \sim S_{21} \supset^l S_{11} : \lfloor S_{11} \rfloor \qquad (*)$$

  By weakening (Lemma 4.30), we can see $\Delta, x{:}S_{21} \vdash S_{21}$ and $\Delta, x{:}S_{21} \vdash S_{11}$. We can reapply the IH to show $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash \psi^l(S_{21}, S_{11}) \sim S_{21} \supset^l S_{11} : \lfloor S_{11} \rfloor$. Now $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l : S_{21} \to S_{11}$ and $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l\, x : S_{11}$. By Lemma 5.8, we can substitute this last into $\Delta, x{:}S_{11} \vdash S_{12}$, finding $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}$.

  We apply the IH for proposition (2) on $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}$ and $\Delta, x{:}S_{21} \vdash S_{22}$, showing

  $$\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22}) \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\} \supset^l S_{22} : \lfloor S_{22} \rfloor \qquad (**)$$

  We now combine (*) and (**) to show $\lfloor \Delta \rfloor \vdash \psi^l(x{:}S_{11} \to S_{12}, x{:}S_{22} \to S_{22}) \sim x{:}S_{11} \to S_{12} \supset^l x{:}S_{11} \to S_{22} : \lfloor S_2 \rfloor$. Let $\lfloor \Delta \rfloor \models \delta$. We can apply (*) to see $\delta_1(\psi^l(S_{21}, S_{11})) \sim \delta_2(S_{21}) \supset^l \delta_2(S_{11}) : \lfloor S_{11} \rfloor$. For the codomain we must show, for all $t \sim s : \lfloor S_{11} \rfloor$, that

  $$\delta_1(\psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22}))\{x := t\} \sim \delta_2(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, s\} \supset^l \delta_2(S_{22})\{x := s\} : \lfloor S_{22} \rfloor$$

  Let $t \sim s : \lfloor S_{11} \rfloor$. Recalling that $\lfloor S_{11} \rfloor = \lfloor S_{21} \rfloor$, observe $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \models \delta\{x := t, s\}$. Call this δ'. By (**) we see

  $$\delta'_1(\psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22})) \sim \delta'_2(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}) \supset^l \delta'_2(S_{22}) : \lfloor S_{22} \rfloor$$

  which we can rewrite to

  $$\delta_1(\psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22}))\{x := t\} \sim \delta_2(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, s\} \supset^l \delta_2(S_{22})\{x := s\} : \lfloor S_{22} \rfloor$$

  This is exactly what we needed to finish the proof of correspondence. □

As a preliminary to type-preservation, we use behavioral correspondence to show that the implication judgment is preserved.

5.10 Lemma: If

- $\emptyset \vdash s_1 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$,
- $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$, and
- $\emptyset \vdash s_1 \supset s_2$

then $\vdash \psi(s_1) \supset \psi(s_2)$.

Proof: By the logical relation. Suppose $\psi(s_1) \longrightarrow_c^* \mathsf{true}$. Theorem 5.9 gives us $\psi(s_1) \sim s_1 : \mathsf{Bool}$, so $s_1 \longrightarrow_h^* \mathsf{true}$ as well. This means that $s_2 \longrightarrow_h^* \mathsf{true}$, by assumption. $\psi(s_2) \sim s_2 : \mathsf{Bool}$ by Theorem 5.9 again, so $\psi(s_2) \longrightarrow_c^* \mathsf{true}$. □

The type preservation proof is very similar to the correspondence proof of Theorem 5.9.

Again, we can only prove that ψ is type preserving after we've proven the behavioral correspondence, since the latter is used to prove that $\psi(\langle\{x{:}B \mid s_1\}, s_2, k\rangle^l) = \langle\{x{:}B \mid \psi(s_1)\}, \psi(s_2), k\rangle^l$ is type preserving. We use behavioral correspondence to show that $\emptyset \vdash s_2 \supset s_1\{x := k\}$ implies $\vdash \psi(s_2) \supset \psi(s_1)\{x := k\}$.

5.11 Theorem [Type preservation for ψ]: 1. If $\psi(s) = t$ and $\Delta \vdash s : S$ then $\lfloor\Delta\rfloor \vdash t : \lfloor S\rfloor$

2. If $\psi^l(S_1, S_2) = c$ and $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor = T$, then $\lfloor\Delta\rfloor \vdash_c c : T$.

Proof: By induction on the lexicographically ordered pair containing (a) the depth of the recursion of the translation ψ or $\psi(s)$, and (b) $|\Delta \vdash s : S|$ or $|\Delta \vdash S_1| + |\Delta \vdash S_2|$.

Part (1) of the proof proceeds by case analysis on the final rule of $\Delta \vdash s : S$, which determines the shape of $\psi(s) = t$ in all cases but S_Sub.

S_Var: $\Delta \vdash x : S$ and $\psi(x) = x$. By inversion $x{:}S \in \Delta$, so $x{:}\lfloor S\rfloor \in \lfloor\Delta\rfloor$. By T_Var, $\lfloor\Delta\rfloor \vdash x : \lfloor S\rfloor$.

S_Const: $\Delta \vdash k : \mathrm{ty_h}(k)$ and $\psi(k) = k$. We've already assumed that the type-assignment functions agree, so $\lfloor \mathrm{ty_h}(k)\rfloor = \mathrm{ty_c}(k)$. By T_Const, $\lfloor\Delta\rfloor \vdash k : \lfloor \mathrm{ty_h}(k)\rfloor$.

S_Lam: $\Delta \vdash \lambda x{:}S_1.\ s_{12} : (x{:}S_1 \to S_2)$ and $\psi(\lambda x{:}S_1.\ s_{12}) = \lambda x{:}\lfloor S_1\rfloor.\ \psi(s_{12})$. By inversion $\Delta, x{:}S_1 \vdash s_{12} : S_2$.

The IH shows $\lfloor\Delta\rfloor, x{:}\lfloor S_1\rfloor \vdash \psi(s_{12}) : \lfloor S_2\rfloor$. By T_Lam, $\lfloor\Delta\rfloor \vdash \lambda x{:}\lfloor S_1\rfloor.\ \psi(s_{12}) : \lfloor S_1\rfloor \to \lfloor S_2\rfloor$.

S_App: $\Delta \vdash s_1\, s_2 : S_2\{x := s_2\}$ and $\psi(s_1\, s_2) = \psi(s_1)\, \psi(s_2)$. By inversion, $\Delta \vdash s_1 : (x{:}S_1 \to S_2)$ and $\Delta \vdash s_2 : S_1$.

By the IH, $\lfloor\Delta\rfloor \vdash \psi(s_1) : \lfloor S_1\rfloor \to \lfloor S_2\rfloor$ and $\lfloor\Delta\rfloor \vdash \psi(s_2) : \lfloor S_1\rfloor$. By T_App, $\lfloor\Delta\rfloor \vdash \psi(s_1\, s_2) : \lfloor S_2\rfloor$.

S_Blame: $\Delta \vdash \Uparrow l : S$ and $\psi(\Uparrow l) = \Uparrow l$. By T_Blame, $\lfloor\Delta\rfloor \vdash \Uparrow l : \lfloor S\rfloor$.

S_Cast: $\Delta \vdash \langle S_1 \Rightarrow S_2\rangle^l : S_1 \to S_2$ and $\psi(\langle S_1 \Rightarrow S_2\rangle^l) = \langle \psi^l(S_1, S_2)\rangle^{l,l}$. By inversion, $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor$.

By the IH for proposition (2), $\lfloor\Delta\rfloor \vdash_c \psi^l(S_1, S_2) : \lfloor S_2\rfloor$. Recalling $\lfloor S_1\rfloor = \lfloor S_2\rfloor$, by T_Contract, $\lfloor\Delta\rfloor \vdash \langle \psi^l(S_1, S_2)\rangle^{l,l} : \lfloor S_1\rfloor \to \lfloor S_2\rfloor$.

S_Checking: $\emptyset \vdash \langle\{x{:}B \mid s_1\}, s_2, k\rangle^l : \{x{:}B \mid s_1\}$ and $\psi(\langle\{x{:}B \mid s_1\}, s_2, k\rangle^l) = \langle\{x{:}B \mid \psi(s_1)\}, \psi(s_2), k\rangle^l$. By inversion we have $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\}$, so $k \in \mathcal{K}_B$. We must reconstruct several premises: typings for the contract, check, and k, as well as the implication.

By inversion of $\emptyset \vdash \{x{:}B \mid s_1\}$, we have $x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$. By the IH, $x{:}B \vdash \psi(s_1) : \mathsf{Bool}$. By T_BaseC, $\emptyset \vdash_c \{x{:}B \mid \psi(s_1)\} : B¿$

By the IH on $\psi(s_2)$ and $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$, we have $\emptyset \vdash \psi(s_2) : \mathsf{Bool}$.

We have by assumption $\lfloor \mathrm{ty_h}(k)\rfloor = \mathrm{ty_c}(k)$, so $\emptyset \vdash k : B$ by T_Const.

Finally, by Lemma 5.10, we can see $\vdash \psi(s_2) \supset \psi(s_1)\{x := k\}$.

S_Sub: $\Delta \vdash s : S$; we do not know anything about the shape of $\psi(s)$. By inversion, $\Delta \vdash s : S'$ and $\Delta \vdash S' <: S$. By Lemma 5.6, $\lfloor S' \rfloor = \lfloor S \rfloor$.

Since the sub-derivation is smaller, by the IH $\lfloor \Delta \rfloor \vdash \psi(s) : \lfloor S' \rfloor$. But $\lfloor S' \rfloor = \lfloor S \rfloor$, so we're done.

Part (2) of the proof proceeds by case analysis on $\psi^l(S_1, S_2) = c$.

- $\psi^l(S_1, \{x{:}B \mid s_2\}) = \{x{:}B \mid \psi(s_2)\}$, so $S_2 = \{x{:}B \mid s_2\}$. By inversion of $\Delta \vdash \{x{:}B \mid s_2\}$, we have $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$.

  By the IH for proposition (1), $\lfloor \Delta \rfloor, x{:}B \vdash \psi(s_2) : \mathsf{Bool}$. By T_BaseC, $\lfloor \Delta \rfloor \vdash_c \{x{:}B \mid \psi(s_2)\} : B$.

- $\psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) = x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22})$, so $S_2 = x{:}S_{21} \to S_{22}$ and $S_1 = x{:}S_{11} \to S_{12}$, where $\lfloor S_{21} \rfloor = \lfloor S_{11} \rfloor$ and $\lfloor S_{22} \rfloor = \lfloor S_{12} \rfloor$. By inversion, we have the following well-formedness derivations:

$$\begin{array}{rr} \Delta \vdash S_{21} & \Delta \vdash S_{11} \\ \Delta, x{:}S_{21} \vdash S_{22} & \Delta, x{:}S_{22} \vdash S_{12} \end{array}$$

  By the IH

$$\lfloor \Delta \rfloor \vdash_c \psi^l(S_{21}, S_{11}) : \lfloor S_{11} \rfloor$$

  Note that $\lfloor \psi^l(S_{21}, S_{11}) \rfloor = \lfloor S_{21} \rfloor$.

  By weakening, we can see $\Delta, x{:}S_{21} \vdash S_{21}$ and $\Delta, x{:}S_{21} \vdash S_{11}$. We can reapply the IH to show $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash_c \psi^l(S_{21}, S_{11}) : \lfloor S_{11} \rfloor$. Now $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l : S_{21} \to S_{11}$. Next $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l\, x : S_{11}$. By Lemma 5.8, we can substitute this last into $\Delta, x{:}S_{11} \vdash S_{12}$, finding $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}$.

  By the IH for proposition (2) on $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}$ and $\Delta, x{:}S_{21} \vdash S_{22}$,

$$\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash_c \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22}) : \lfloor S_{22} \rfloor$$

  By Lemma 5.7, $\lfloor \psi^l(S_{21}, S_{11}) \rfloor = \lfloor S_{21} \rfloor$, so we can rewrite the above derivation to

$$\lfloor \Delta \rfloor, x{:}\psi^l(S_{21}, S_{11}) \vdash_c \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22}) : \lfloor S_{22} \rfloor$$

  Now by T_FunC

$$\lfloor \Delta \rfloor \vdash_c x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, x\}, S_{22}) : \lfloor S_{21} \rfloor \to \lfloor S_{22} \rfloor$$

□

Terms

$$\begin{array}{rcl}
\phi(\Gamma_1, x{:}T\,,\, \Gamma_2 \vdash x : T) & = & x \\
\phi(\Gamma_1, x{:}c\,,\, \Gamma_2 \vdash x : \lfloor c \rfloor) & = & \langle \phi(\Gamma_1 \vdash_c c : \lfloor c \rfloor) \Rightarrow \lceil c \rceil \rangle^{l_0}\, x \\
\phi(\Gamma \vdash k : T) & = & k \\
\phi(\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2) & = & \lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_2 : T_2) \\
\phi(\Gamma \vdash t_1\, t_2 : T_2) & = & \phi(\Gamma \vdash t_1 : T_1 \to T_2)\, \phi(\Gamma \vdash t_2 : T_1) \\
\phi(\Gamma \vdash \Uparrow l : T) & = & \Uparrow l \\
\phi(\emptyset \vdash \langle c, t, k \rangle^l : B) & = & \langle \phi(\emptyset \vdash_c c : B), \phi(\emptyset \vdash t : B), k \rangle^l \\
\phi(\Gamma \vdash \langle c \rangle^{l,l'} : T) & = & \lambda x{:}\lceil c \rceil.\ \langle \phi(\Gamma \vdash_c c : T) \Rightarrow \lceil c \rceil \rangle^{l'}\, (\langle \lceil c \rceil \Rightarrow \phi(\Gamma \vdash_c c : T) \rangle^{l}\, x) \qquad \text{where } x \text{ is fresh}
\end{array}$$

Types

$$\begin{array}{rcl}
\phi(\Gamma \vdash_c \{x{:}B \mid t\} : B) & = & \{x{:}B \mid \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})\} \\
\phi(\Gamma \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2) & = & x{:}\phi(\Gamma \vdash_c c_1 : T_1) \to \phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)
\end{array}$$

Contexts

$$\begin{array}{rcl}
\phi(\vdash \emptyset) & = & \emptyset \\
\phi(\vdash \Gamma, x{:}T) & = & \phi(\vdash \Gamma), x{:}\lceil T \rceil \\
\phi(\vdash \Gamma, x{:}c) & = & \phi(\vdash \Gamma), x{:}\phi(\Gamma \vdash_c c : \lfloor c \rfloor)
\end{array}$$

Figure 22: The translation ϕ from dependent λ_{C} to dependent λ_{H}

6 Translating λ_{C} to λ_{H}: dependent ϕ

Things get more interesting when we consider the translation ϕ from dependent λ_{C} to dependent λ_{H}. We can prove that it is type preserving (for terms without active checks), but we can only show a weaker behavioral correspondence: sometimes λ_{C} terms terminate with values when their ϕ-images go to blame. This weaker property is a consequence of the asymmetrically substituting F_CDecomp rule and extra casts inserted for type preservation.

The full definition is in Figure 22. One point to note is that, in the dependent case, we need to translate *derivations* of well-formedness and well-typing of λ_{C} contexts, terms, and contracts into λ_{H} contexts, terms, and types. We need to use derivations to ensure type preservation. Notice that T_VarT and T_VarC derivations are translated differently: variables bound to simple types translate to themselves, but we add a cast (with distinguished label l_0) to variables bound to contracts.

To see why we make this distinction, consider the function contract $f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\,0\ =\ 0\}$. Note that this contract is well-formed in λ_{C}, but that the codomain "abuses" the bound variable. A naïve translation will *not* be well-typed in λ_{H}, since $f\,0$ will not be typeable in the context $f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\}), z{:}\lceil \mathsf{Int} \rceil$—$f$ only accepts positive arguments. The problem is that WF_Fun can add a non-raw type to the context, so we need to restore the "variables have raw types" invariant—something we can't always rely on subtyping to do. By tracking which variables were bound by contracts in λ_{C}, we can be sure to cast them to raw types when they're referenced—this motivates the $x{:}c$ binding form in dependent λ_{C}. We therefore translate the contract above to $f{:}S \to \{z{:}\mathsf{Int} \mid (\langle S \Rightarrow \lceil \mathsf{Int} \to \mathsf{Int} \rceil \rangle^{l_0} f)\,0\ =\ 0\}$, where $S = x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\}$.

Just like in the nondependent case in Section 3, bulletproofing uses raw types, which are defined in Section 3. We redefine them here for the dependent system.

$$\begin{array}{rclcrcl}
\lceil \{x{:}B \mid s\} \rceil & = & \{x{:}B \mid \mathsf{true}\} & \qquad & \lceil x{:}S_1 \to S_2 \rceil & = & \lceil S_1 \rceil \to \lceil S_2 \rceil \\
\lceil B \rceil & = & \{x{:}B \mid \mathsf{true}\} & & \lceil T_1 \to T_2 \rceil & = & \lceil T_1 \rceil \to \lceil T_2 \rceil \\
\lceil \{x{:}B \mid t\} \rceil & = & \{x{:}B \mid \mathsf{true}\} & & \lceil x{:}c_1 \mapsto c_2 \rceil & = & \lceil c_1 \rceil \to \lceil c_2 \rceil
\end{array}$$

Note that x is unreferenced, and can vary without affecting $\lceil S_2 \rceil$ or the well-formedness of the type

We could write the translation on terms instead of derivations, defining

$$\phi(x{:}c_1 \mapsto c_2) = x{:}\phi(c_1) \to \phi(c_2)\{x := \langle \phi(c_1) \Rightarrow \lceil c_1 \rceil \rangle^l\, x\}$$

but the proofs are easier if we translate derivations.

Constants translate to themselves. One technical point is that, to maintain the raw type invariant, we need λ_{H}'s higher-order constants to have typings that can be seen as raw by the subtyping relation, i.e.,

$\Delta \vdash \mathrm{ty_h}(k) <: \lceil \mathrm{ty_c}(k) \rceil$. This can be proven at base types (since we have already assumed that $\mathrm{ty_h}(k)$ is the "most specific type" for each k), but must be assumed for higher-order types.This slightly restricts the types we might assign to our constants, e.g., we cannot say $\mathrm{ty_h}(\mathsf{sqrt}) = x{:}\{x{:}\mathsf{Float} \mid x \; = \; 0\} \rightarrow \{y{:}\mathsf{Float} \mid (y \; * \; y) \; = \; x\}$, since it is not the case that $\Delta \vdash \mathrm{ty_h}(\mathsf{sqrt}) <: \lceil \mathsf{Float} \rightarrow \mathsf{Float} \rceil$. Since its domain cannot be refined, $[\![\mathsf{sqrt}]\!]$ must be defined for all $k \in \mathcal{K}_{\mathsf{Float}}$, e.g., $[\![\mathsf{sqrt}]\!](-1)$ must be defined. We've already required that denotations be total over their simple types in λ_{C}, and λ_{H} uses the same denotation function $[\![-]\!]$, so the subtyping requirement does not seem too severe. In any case, we can define it to be equal to $\Uparrow l_0$, for some distinguished label l_0.We could instead translate k to $\langle \mathrm{ty_h}(k) \Rightarrow \lceil \mathrm{ty_h}(k) \rceil \rangle^{l_0} \; k$; however, in this case the nondependent fragments of the languages would no longer correspond exactly.

We show the behavioral correspondence using a blame-inexact logical relation, defined in Figure 23. We read $\Gamma \vdash t \sim_{\succ} s : T$ as "Γ shows t corresponds to s at T". The behavioral correspondence here, though weaker than we had before, is still pretty strong: if $t \sim_{\succ} s : B$, then either $s \longrightarrow_h^* \Uparrow l$ or both t and s go to $k \in \mathcal{K}_B$. As in Section 5, we use the term correspondence to define a correspondence relating contracts and λ_{H} types, and then we lift both correspondences to open terms with dual closing substitions, also defined in Figure 23. Closing substitutions map variables to terms corresponding at appropriate type. Note that closing substitutions treat the $x{:}c$ bindings in the context Γ as if they were $x{:}T$.

We make assumptions and prove lemmas to similar those in Section 6.

6.1 Lemma [Raw types are well formed]: For all Δ, S, c, and T, we have:

$\Delta \vdash \lceil S \rceil$,
$\Delta \vdash \lceil c \rceil$, and
$\Delta \vdash \lceil T \rceil$.

Proof: By three separate inductions: two on the structure of c and T, and one on $|S|$. □

6.2 Lemma [Skeletal equality for translated contracts]: $\lfloor \phi(\Gamma \vdash_c c : T) \rfloor = \lfloor \lceil \phi(\Gamma \vdash_c c : T) \rceil \rfloor = T$ for all T, c, and σ.

Proof: By induction on the contract typing derivation.

T_RefineC: Immediate.

T_FunC: By the IH. □

6.3 Lemma [Preservation of contract / type-agreement]: $\lceil \phi(\Gamma \vdash_c c : T) \rceil = \lceil T \rceil$.

Proof: By induction on the contract typing derivation.

T_RefineC: Immediate.

T_FunC: By the IH. □

Some standard facts about the logical relation.

6.4 Lemma [Expansion and contraction]: If:

- $t \longrightarrow_c^* t'$, and
- $s \longrightarrow_h^* s'$

then $t \sim_{\succ} s : T$ iff $t' \sim_{\succ} s' : T$.

Proof: By coevaluation for $\longrightarrow_c^*$ (Corollary 4.3)and $\longrightarrow_h^*$ (Corollary 4.8), both t and $t' \longrightarrow_c^* r$, likewise s and $s' \longrightarrow_h^* q$, and we have $r \approx_{\succ} q : T$. □

Note that there are corresponding terms at every type. We prove a much stronger lemma than we did for $\sim$ in Lemma 5.2, since the correspondence is much weaker.

6.5 Lemma [Everything is corresponds to blame]: For all t and T, $t \sim_{\succ} \Uparrow l' : T$.

Proof: By definition, $t \longrightarrow_c^* r$. $r \approx_{\succ} \Uparrow l' : B$ and $r \approx_{\succ} \Uparrow l' : T_1 \to T_2$ by definition. ☐

6.6 Lemma [Constants correspond to themselves]: For all k, $k \approx_{\succ} k : \mathrm{ty_c}(k)$.

Proof: By induction on $\mathrm{ty_c}(k)$.

$\underline{\mathrm{ty_c}(k) = B}$: By definition.

$\underline{\mathrm{ty_c}(k) = B_1 \to B_2 \to \ldots \to B_n}$: We wish to show that $k \approx_{\succ} k : B_1 \to B_2 \to \ldots \to B_n$. Let $t \sim_{\succ} s : B_1$. By definition, either $s \longrightarrow_h^* \Uparrow l$ or $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ such that $v \approx w : B_1$. In the latter case, $k \Uparrow l \longrightarrow_h^* \Uparrow l$, and we are done by Lemma 6.5.

Note that the only values related at B_1 are constants, so $v = w = k' \in \mathcal{K}_{B_1}$. Since denotations are total, $[\![k]\!](k')$ is defined. Moreover, $[\![k]\!](k')$ is either a constant k'' or $\Uparrow l$, by definition. In the latter case the applications are related because λ_{H} goes to blame. In the former, $\mathrm{ty_c}(k'') = B_2 \to \ldots \to B_n$, so $k'' \approx_{\succ} k'' : B_2 \to \ldots \to B_n$ by the IH. ☐

6.7 Lemma [Coevaluation modulo blame]: If $t \sim_{\succ} s : T$, then either $s \longrightarrow_h^* \Uparrow l$ or $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ (where $v \approx_{\succ} w : T$).

Proof: By induction on T. We have $t \longrightarrow_c^* r$ and $s \longrightarrow_h^* q$ such that $r \approx_{\succ} q : T$ by definition.

- $T = B$. There are two cases:
  - $r = q = k \in \mathcal{K}_B$; in this case, t and s go to values, and $k \approx_{\succ} k : B$ by definition.
  - $q = \Uparrow l$. In this case, $s \longrightarrow_h^* \Uparrow l$.
- $T = T_1 \to T_2$. Again, there are two cases:
  - $r = v$, $q = w$, and $\forall t' \sim_{\succ} s' : T_1.\ v\,t \sim_{\succ} w\,s : T_2$. Here t and s go to corresponding values.
  - $q = \Uparrow l$. Here, $s \longrightarrow_h^* \Uparrow l$. ☐

As a corollary of Lemma 5.2 and Lemma 6.4, if two terms evaluate to blame—or even just the λ_{H} side!—then they correspond. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

We prove three lemmas about contracts and casts at base types. The first two characterize contracts and casts at base types. The third shows that correspondence is closed under adding extra casts to the λ_{H} term, due to the inexactness of our behavioral correspondence.

6.8 Lemma [Trivial casts]: If $t \sim_{\succ} s : B$, then $t \sim_{\succ} \langle S \Rightarrow \lceil B \rceil \rangle^l\, s : B$ for all S.

Proof: By coevaluation (Lemma 6.7), either $s \longrightarrow_h^* \Uparrow l'$ or s and t both reduce to $k \in \mathcal{K}_B$. In the former case, $\langle S \Rightarrow \lceil B \rceil \rangle^l\, s \longrightarrow_h^* \Uparrow l'$ and we are done by Lemma 6.5. In the case when both go to k, $\langle S \Rightarrow \lceil B \rceil \rangle^l\, s \longrightarrow_h^* \langle \{x{:}B \mid \mathsf{true}\}, \mathsf{true}, k \rangle^l$, which steps to k, and $k \approx_{\succ} k : B$ by definition. ☐

6.9 Lemma [Related base casts]: If $\{x{:}B \mid t\} \sim_{\succ} \{x{:}B \mid s\} : B$ and $t' \sim_{\succ} s' : B$, then $\langle \{x{:}B \mid t\} \rangle^{l,l'}\, t' \sim_{\succ} \langle S \Rightarrow \{x{:}B \mid s\} \rangle^l\, s' : B$ for all S.

Proof: Direct. By coevaluation (Lemma 6.7), either $s' \longrightarrow_h^* \Uparrow l''$ or s' and t' both reduce to $k \in \mathcal{K}_B$. In the former case, $\langle S \Rightarrow \{x{:}B \mid s\} \rangle^l\, s' \longrightarrow_h^* \Uparrow l''$ and we are done by Lemma 6.5.

In the case when both go to k, $\langle \{x{:}B \mid t\} \rangle^{l,l'}\, t' \longrightarrow_c^* \langle \{x{:}B \mid t\}, t\{x := k\}, k \rangle^l$ and $\langle S \Rightarrow \{x{:}B \mid s\} \rangle^l\, s' \longrightarrow_h^* \langle \{x{:}B \mid s\}, s\{x := k\}, k \rangle^l$. We know that $t\{x := k\} \sim_{\succ} s\{x := k\} : \mathsf{Bool}$, so either $s\{x := k\} \longrightarrow_h^* \Uparrow l''$ (which means we're done by Lemma 6.5) or both go to the same boolean.

If they both go to false, then both go to $\Uparrow l$, and $\Uparrow l \approx_{\succ} \Uparrow l : B$. If they both go to true, then both go to k and $k \approx_{\succ} k : B$. In either case, $\langle \{x{:}B \mid t\} \rangle^{l,l'}\, t' \sim_{\succ} \langle S \Rightarrow \{x{:}B \mid s\} \rangle^l\, s' : B$ by expansion (Lemma 6.4). ☐

Since $\lambda_{\rm H}$ terms can go to blame more often than corresponding $\lambda_{\rm C}$ terms, we can always add "extra" casts to $\lambda_{\rm H}$ terms. We formalize this in the following lemma, which captures the asymmetric treatment of blame by the $\sim_{\succ}$ relation. This lemma is crucial in the presence of the asymmetric F_CDecomp rule—we use it to show that the cast substituted in the codomain does not affect behavioral correspondence. Note that the statement of the lemma requires that the types of the cast correspond to *some* contracts at the same type T, but we never use the contracts in the proof—they're witnesses to the well-formedness of the $\lambda_{\rm H}$ types.

6.10 Lemma [Extra casts]: If $t \sim_{\succ} s : T$ and $c_1 \sim_{\succ} S_1 : T$ and $c_2 \sim_{\succ} S_2 : T$, then $t \sim_{\succ} \langle S_1 \Rightarrow S_2\rangle^l\, s : T$.

Proof: The proof is by induction on T. Note that we do not use c_1 or c_2 at all in the proof, but instead they are witnesses to the well-formedness of S_1 and S_2.

$\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = T$. By coevaluation (Lemma 6.7), either $s \longrightarrow_h^* \Uparrow l'$ or t and s both go to values related at T. If $s \longrightarrow_h^* \Uparrow l'$, then $\langle S_1 \Rightarrow S_2\rangle^l\, s \longrightarrow_h^* \Uparrow l'$ and $t \sim_{\succ} \Uparrow l' : T$ since everything is related to blame (Lemma 6.5).

Therefore, suppose that $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ and $v \approx_{\succ} w : T$ in each of the following cases of the induction.

- $T = B$, $S_2 = \{x{:}B \mid s_2\}$, and $c_2 = \{x{:}B \mid t_2\}$.

  So $t \longrightarrow_c^* k$ and $s \longrightarrow_h^* k$ for $k \in \mathcal{K}_B$. If t and s both go to k, then $\langle S_1 \Rightarrow S_2\rangle^l\, s \longrightarrow_h^* \langle\{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l$. By $c_2 \sim_{\succ} S_2 : B$ we see (in particular) $t_2\{x := k\} \sim_{\succ} s_2\{x := k\} : \mathsf{Bool}$. So $s_2\{x := k\}$ either goes to $\Uparrow l'$ or $s_2\{x := k\}$ (and, irrelevantly, $t_2\{x := k\}$) go to some $k' \in \mathcal{K}_{\mathsf{Bool}}$ (by Lemma 6.7). In the former case, $\langle\{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l \longrightarrow_h^* \Uparrow l'$ and we are done (Lemma 6.5). In the latter case, the $\lambda_{\rm H}$ term either goes to $\Uparrow l$ (and everything is related to blame) or goes to k—but so does t, and $k \approx_{\succ} k : B$.

- $T = T_1 \to T_2$. We have:

  $$\begin{array}{rclcrcl} S_1 &=& x{:}S_{11} \to S_{12} & \quad & S_2 &=& x{:}S_{21} \to S_{22} \\ c_1 &=& x{:}c_{11} \mapsto c_{12} & & c_2 &=& x{:}c_{21} \mapsto c_{22} \end{array}$$

  We have $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$, where $v \approx_{\succ} w : T_1 \to T_2$.

  Let $t' \sim_{\succ} s' : T_1$; we wish to see that $v\,t' \sim_{\succ} (\langle S_1 \Rightarrow S_2\rangle^l\, w)\, s' : T_2$. By coevaluation (Lemma 6.7), either $s' \longrightarrow_h^* \Uparrow l'$ or both go to values. In the former case the whole cast goes to $\Uparrow l'$ we are done by Lemma 6.5, so let $t' \longrightarrow_c^* v'$ and $s' \longrightarrow_h^* w'$.

  Decomposing the cast in $\lambda_{\rm H}$,

  $$(\langle S_1 \Rightarrow S_2\rangle^l\, w)\, s' \longrightarrow_h^* \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow S_{22}\{x := w'\}\rangle^l\, (w\, (\langle S_{21} \Rightarrow S_{11}\rangle^l\, w'))$$

  We have $c_{21} \sim_{\succ} S_{21} : T_1$ and $c_{11} \sim_{\succ} S_{11} : T_1$, so $v' \sim_{\succ} \langle S_{21} \Rightarrow S_{11}\rangle^l\, w' : T_1$ by the IH. Since $v \approx_{\succ} w : T_1 \to T_2$, we can see that $v\,v' \sim_{\succ} w\,(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w') : T_2$.

  Furthermore, we know that for all $t'' \sim_{\succ} s'' : T_1$ that

  - $c_{12}\{x := t''\} \sim_{\succ} S_{12}\{x := s''\} : T_2$ and
  - $c_{22}\{x := t''\} \sim_{\succ} S_{22}\{x := s''\} : T_2$.

  We know that $v' \sim_{\succ} w' : T_1$ and $v' \sim_{\succ} \langle S_{21} \Rightarrow S_{11}\rangle^l\, w' : T_1$, so we can see

  - $c_{12}\{x := v'\} \sim_{\succ} S_{12}\{x := w'\} : T_2$ and
  - $c_{22}\{x := v'\} \sim_{\succ} S_{22}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} : T_2$.

So by the IH,

$$v\, v' \sim_{\succ} \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\} \Rightarrow S_{22}\{x := w'\}\rangle^l\, (w\, (\langle S_{21} \Rightarrow S_{11}\rangle^l\, w')) \,:\, T_2$$

and we are done by expansion (Lemma 6.4). □

To apply the extra cast lemma, we'll need these "witness" contracts for raw types; to that end we define trivial contracts. These contracts are *lifted* from types, and are the λ_{C} correlate to λ_{H}'s raw types.

$$\begin{array}{rcl} B\uparrow & = & \{x{:}B \mid \mathsf{true}\} \\ T_1 \to T_2\uparrow & = & T_1\uparrow \mapsto T_2\uparrow \end{array}$$

6.11 Lemma [Lifted types are well formed]: For all T and Γ, $\Gamma \vdash_c T\uparrow : T$.

Proof: By induction on T.

- $T = B$. By T_Const, observing that $\Gamma, x{:}B \vdash \mathsf{true} : \mathsf{Bool}$.
- $T = T_1 \to T_2$. By the IH, $\Gamma \vdash_c T_1\uparrow : T_1$ and $\Gamma, x{:}T_1\uparrow \vdash_c T_2\uparrow : T_2$. By T_FunC, $\Gamma \vdash_c (T_1 \to T_2)\uparrow : T_1 \to T_2$. □

6.12 Lemma [Lifted types logically relate to raw types]: For all T, $T\uparrow \sim_{\succ} \lceil T \rceil : T$.

Proof: By induction on T.

- $T = B$. Immediate, since $\mathsf{true} \approx_{\succ} \mathsf{true} : \mathsf{Bool}$.
- $T = T_1 \to T_2$. By the IHs (the substitution does nothing). □

The "bulletproofing" lemma is the "key" to the behavioral correspondence proof. We show that a contract application corresponds to bulletproofing with related types. Note that we allow for different types in the two casts. This is necessary due to an asymmetric substitution that occurs in the case when $T = B \to T_2$.

6.13 Lemma [Bulletproofing]: If $t \sim_{\succ} s : T$ and $c \sim_{\succ} S : T$ and $c \sim_{\succ} S' : T$, then $\langle c\rangle^{l,l'}\, t \sim_{\succ} \langle S' \Rightarrow \lceil S' \rceil\rangle^{l'}\, \langle \lceil S \rceil \Rightarrow S\rangle^l\, s : T$.

Proof: By induction on T. First, observe that (by coevaluation, Lemma 6.7) either $s \longrightarrow_h^* \Uparrow l''$ or both t and s go to values related at T. In the former case, $\langle S' \Rightarrow \lceil S' \rceil\rangle^{l'}\, \langle \lceil S \rceil \Rightarrow S\rangle^l\, s \longrightarrow_h^* \Uparrow l''$, and everything is related to blame (Lemma 6.5). So $t \longrightarrow_c^* v$, $s \longrightarrow_h^* w$, and $v \approx_{\succ} w : T$.

- $T = B$. So $c = \{x{:}B \mid t_1\}$ and $S = \{x{:}B \mid s_1\}$ and $S' = \{x{:}B \mid s_2\}$. By Lemma 6.9 we have $\langle c\rangle^{l,l'}\, t \sim_{\succ} \langle \lceil S \rceil \Rightarrow S\rangle^l\, s : B$ By Lemma 6.8 we can add the extra cast.
- $T = T_1 \to T_2$. We know that $c = x{:}c_1 \mapsto c_2$, $S = x{:}S_1 \to S_2$ and $S' = x{:}S_1' \to S_2'$. Let $t' \sim_{\succ} s' : T_1$. By Lemma 6.7 and Lemma 6.5, we only need to consider the case where $t' \longrightarrow_c^* v'$ and $s' \longrightarrow_h^* w'$—if $s' \longrightarrow_h^* \Uparrow l''$ we're done.

  On the λ_{C} side, $(\langle c\rangle^{l,l'}\, t)\, t' \longrightarrow_c^* \langle c_2\{x := v'\}\rangle^{l,l'}\, (v\, (\langle c_1\rangle^{l',l}\, v'))$. In λ_{H}, we can see

  $$\begin{array}{l} (\langle S' \Rightarrow \lceil S' \rceil\rangle^{l'}\, \langle \lceil S \rceil \Rightarrow S\rangle^l\, s)\, s' \longrightarrow_h^* \\ \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1'\rangle^{l'}\, w'\} \Rightarrow \lceil S_2' \rceil\rangle^{l'}\, ((\langle \lceil S \rceil \Rightarrow S\rangle^l\, w)\, (\langle \lceil S_1' \rceil \Rightarrow S_1'\rangle^{l'}\, w')) \end{array}$$

  We cannot determine where the redex is until we know the shape of T_1—does the negative argument cast step to an active check, or do we decompose the positive cast?

– $T_1 = B$.
By Lemma 6.9 and $c_1 \sim_{\succ} S_1' : B$, we know that $\langle c_1 \rangle^{l',l} v' \sim_{\succ} \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w' : B$. By coevaluation (Lemma 6.7), the λ_{H} term goes to blame or they both go to the same value, $v' = w' = k \in \mathcal{K}_B$. In the former case, then the entire λ_{H} term goes to blame and we are done by Lemma 6.5. So suppose $\langle c_1 \rangle^{l',l} k \longrightarrow_c^* k$ and $\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w' \longrightarrow_h^* k$. Now the terms evaluate like so:

$$\langle c_2\{x := v'\} \rangle^{l,l'} (v\,(\langle c_1 \rangle^{l',l} v')) \longrightarrow_c^* \langle c_2\{x := k\} \rangle^{l,l'} (v\,k)$$

$$\begin{array}{l}\langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} (((\langle \lceil S \rceil \Rightarrow S \rangle^{l} w)\,(\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w')) \longrightarrow_h^* \\ \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \\ \quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\} \rangle^{l} (w\,(\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l} k))\end{array}$$

By Lemma 6.8, $k \sim_{\succ} \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l} k : B$, so $v\,k \sim_{\succ} w\,(\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l} k) : T_2$.
Noting that $k \sim_{\succ} k : B$ and $k \sim_{\succ} \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l} k : B$, we can see that $c_2\{x := k\} \sim_{\succ} S_2\{x := k\} : T_2$ and $c_2\{x := k\} \sim_{\succ} S_2'\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l} k\} : T_2$. Now the IH shows that $\langle c_2\{x := k\} \rangle^{l,l'} (v\,k) \sim_{\succ} \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\} \rangle^{l} (w\,(\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l} k)) : T_2$, and we conclude the case with expansion (Lemma 6.4).

– $T_1 = T_{11} \to T_{12}$. We can continue with an application of F_CDecomp in λ_{H} and find:

$$\begin{array}{l}\langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} (((\langle \lceil S \rceil \Rightarrow S \rangle^{l} w)\,(\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w')) \longrightarrow_h^* \\ \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \\ \quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} \rangle^{l} \\ \quad\quad (w\,(\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l} \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'))\end{array}$$

By the IH, $\langle c_1 \rangle^{l',l} v' \sim_{\succ} \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l} \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w' : T_1$. By assumption, the results of applying v and w to these values correspond. (And they *are* values, since function contracts/casts applied to values are values.)
We know $c_1 \sim_{\succ} S_1' : T_1$, and by Lemma 6.12 $T_1 \uparrow\sim_{\succ} \lceil S_1' \rceil : T_1$. Since $v' \sim_{\succ} w' : T_1$, Lemma 6.10 shows $v' \sim_{\succ} \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w' : T_1$. This lets us see that $c_2\{x := v'\} \sim_{\succ} S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} : T_2$ and $c_2\{x := v'\} \sim_{\succ} S_2\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'} w'\} : T_2$. Now the IH and expansion (Lemma 6.4) complete the proof. ☐

Having characterized how contracts and pairs of related casts relate, we show that translated terms correspond to their sources.

6.14 Theorem [Behavioral correspondence modulo blame]: Suppose $\vdash \Gamma$.

1. If $\phi(\Gamma \vdash t : T) = s$ then $\Gamma \vdash t \sim_{\succ} s : T$.
2. If $\phi(\Gamma \vdash_c c : T) = S$ then $\Gamma \vdash c \sim_{\succ} S : T$.

Proof: We simultaneously show both properties by induction on the depth of ϕ's recursion. To show $\Gamma \vdash t \sim_{\succ} s : T$, let $\Gamma \models \delta$—we will show $\delta_1(t) \sim_{\succ} \delta_2(s) : T$.

The proof proceeds by case analysis on the final rule of the translated typing and well-formedness derivations.

T_VarT:] $\phi(\Gamma \vdash x : T) = x$. By inversion, $\Gamma = \Gamma_1, x{:}T, \Gamma_2$. We see by assumption that $\delta_1(x) \sim_{\succ} \delta_2(x) : T$.

T_VarC: $\phi(\Gamma \vdash x : \lfloor c \rfloor) = \langle \phi(\Gamma_1 \vdash_c c : T) \Rightarrow \lceil T \rceil \rangle^{l_0} x$, where $\Gamma = \Gamma_1, x{:}c, \Gamma_2$.

By assumption $\delta_1(x) \sim_{\succ} \delta_2(x) : T$. We know by the IH that $\Gamma_1 \vdash c \sim_{\succ} \phi(\Gamma_1 \vdash_c c : T) : T$; since $\Gamma \models \delta$, we also have $\Gamma_1 \models \delta$, so $\delta_1(c) \sim_{\succ} \delta_2(\phi(\Gamma_1 \vdash_c c : T)) : T$.

By Lemma 6.12, $\Gamma_1 \vdash T \uparrow\sim_{\succ} \lceil T \rceil : T$. Next, $\Gamma_1 \models \delta$, since $\Gamma \models \delta$. By Lemma 6.10, $\delta_1(x) \sim_{\succ} \langle \delta_2(\phi(\Gamma_1 \vdash_c c : T)) \Rightarrow \lceil T \rceil \rangle^{l_0} \delta_2(x) : T$.

T_Const: $\phi(\Gamma \vdash k : \mathrm{ty_c}(k)) = k$. We have $k \sim k : \mathrm{ty_c}(k)$ by Lemma 6.6.

T_Lam: $\phi(\Gamma \vdash \lambda x{:}T_1.\ t_{12} : T_1 \to T_2) = \lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2)$. We need to show that

$$\delta_1(\lambda x{:}T_1.\ t_{12}) \sim \delta_2(\lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2)) : T_1 \to T_2$$

Let $t' \sim_{\succ} s' : T_1$; we now need to show

$$\delta_1(\lambda x{:}T_1.\ t_{12})\, t' \sim \delta_2(\lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2))\, s' : T_1 \to T_2$$

By coevaluation (Lemma 6.7), either $s' \longrightarrow_h^* \Uparrow l$ or s' and t' go to related values. In the former case, $(\lambda x{:}\lceil T_1 \rceil.\ \delta_2(\phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2)))\, s' \longrightarrow_h^* \Uparrow l$, and anything is related to blame (Lemma 6.5).

Suppose $t' \longrightarrow_c^* v$ and $s' \longrightarrow_h^* w$, where $v \sim_{\succ} w : T_1$. We can see that $(\lambda x{:}T_1.\ \delta_1(t_{12}))\, t \longrightarrow_c^* \delta_1(t_{12})\{x := v\}$ and $(\lambda x{:}\lceil T_1 \rceil.\ \delta_2(\phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2)))\, s \longrightarrow_h^* \delta_2(\phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2))\{x := w\}$. By the IH, $\Gamma, x{:}T_1 \vdash t_{12} \sim_{\succ} \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2) : T_2$. Since $\Gamma, x{:}T_1 \models \delta\{x := v, w\}$, the IH shows that the F_Beta-reduced terms correspond, so we conclude that the lambdas correspond by expansion (Lemma 6.4).

T_App: $\phi(\Gamma \vdash t_1\, t_2 : T_2) = \phi(\Gamma \vdash t_1 : T_1 \to T_2)\, \phi(\Gamma \vdash t_2 : T_1)$. We must show

$$\delta_1(t_1\, t_2) \sim \delta_2(\phi(\Gamma \vdash t_1 : T_1 \to T_2)\, \phi(\Gamma \vdash t_2 : T_1)) : T_2$$

By the IH, $\delta_1(t_1) \sim_{\succ} \delta_2(\phi(\Gamma \vdash t_1 : T_1 \to T_2)) : T_1 \to T_2$ and $\delta_1(t_2) \sim_{\succ} \delta_2(\phi(\Gamma \vdash t_2 : T_1)) : T_1$.

By coevaluation (Lemma 6.7), either $\delta_2(\phi(\Gamma \vdash t_1 : T_1 \to T_2)) \longrightarrow_h^* \Uparrow l$ or both it and $\delta_1(t_1)$ go to corresponding values.

In the former case, the whole λ_{H} application goes to $\Uparrow l$, and everything corresponds to blame (Lemma 6.5).

Suppose $\delta_1(t_1) \longrightarrow_c^* v_1$ and $\delta_2(\phi(\Gamma \vdash t_1 : T_1 \to T_2)) \longrightarrow_h^* w_1$ such that $v_1 \approx w_1 : T_1 \to T_2$. By the "logicality" of the relation, $v_1\, \delta_1(t_2) \sim_{\succ} w_1\, \delta_2(\phi(\Gamma \vdash t_2 : T_1)) : T_2$. The two original applications correspond by expansion (Lemma 6.4).

T_Blame: $\phi(\Gamma \vdash \Uparrow l : T) = \Uparrow l$. We can see $\delta_1(\Uparrow l) \sim \delta_2(\Uparrow l) : T$ by Lemma 6.5.

T_Contract: $\phi(\Gamma \vdash \langle c \rangle^{l,l'} : T \to T) = \lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'} (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x)$, where $S = \phi(\Gamma \vdash_c c : T)$. We show that

$$\delta_1(\langle c \rangle^{l,l'}) \sim_{\succ} \delta_2(\lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'} (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x)) : T \to T$$

Let $t \sim s : T$; we need to show

$$\delta_1(\langle c \rangle^{l,l'})\, t \sim_{\succ} \delta_2(\lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'} (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x))\, s : T$$

Either $s \longrightarrow_h^* \Uparrow l''$ or t and s coevaluate to values. If s goes to $\Uparrow l''$, then the whole cast does, too, and we are done by Lemma 6.5.

If t and s evaluate to $v \sim w : T$. We can evaluate the λ_{H} term to see:

$$\delta_2(\lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'} (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x))\, s \longrightarrow_h^* \delta_2(\langle S \Rightarrow \lceil S \rceil \rangle^{l'})\, (\delta_2(\langle \lceil S \rceil \Rightarrow S \rangle^{l})\, s)$$

By the IH on $\phi(\Gamma \vdash_c c : T)$, we see $\delta_1(c) \sim_{\succ} \delta_2(S) : T$. By the bulletproofing lemma (Lemma 6.13), we see $\langle \delta_1(c) \rangle^{l,l'}\, v \sim_{\succ} \langle \delta_2(S) \Rightarrow \lceil S \rceil \rangle^{l'} \langle \lceil S \rceil \Rightarrow \delta_2(S) \rangle^{l}\, w : T$.

T_Checking: $\phi(\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B) = \langle \phi(\emptyset \vdash_c \{x{:}B \mid t_1\} : B), \phi(\emptyset \vdash t_2 : \mathsf{Bool}), k \rangle^{l}$. We show that the two are logically related. (Note that we do not need δ, since $\Gamma = \emptyset$.)

By the IH, if $\phi(\emptyset \vdash t_2 : \mathsf{Bool}) = s_2$, we can see that $t_2 \sim_{\succ} s_2 : \mathsf{Bool}$. By definition, $t_2 \longrightarrow_c^* r_2$ and $s_2 \longrightarrow_h^* q_2$ such that $r_2 \approx q_2 : \mathsf{Bool}$. There are three cases:

- $q_2 = \Uparrow l'$. The λ_{H} active check steps to $\Uparrow l'$.
- $r_2 = q_2 = \mathsf{false}$. Both checks step to $\Uparrow l$.
- $r_2 = q_2 = \mathsf{true}$. Both checks step to k. Since $\emptyset \vdash k : B$, we know $k \in \mathcal{K}_B$, and so $k \sim k : B$.

In all three cases, either the λ_{H} term steps to blame or both checks step to related values.

T_RefineC: $\phi(\Gamma \vdash_c \{x{:}B \mid t\} : B) = \{x{:}B \mid \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})\}$. To see that the contract and type correspond, we must show that, for all $k \in \mathcal{K}_B$, $\delta_1(t)\{x := k\} \sim_{\succ} \delta_2(\phi(\Gamma, x{:}B \vdash t : \mathsf{Bool}))\{x := k\} : \mathsf{Bool}$.

By the IH on $\phi(\Gamma \vdash t : \mathsf{Bool}) = s$, we see $\Gamma, x{:}B \vdash t \sim_{\succ} s : \mathsf{Bool}$. Let $k \in \mathcal{K}_B$. It is immediate that $k \sim k : B$, so $\Gamma, x{:}B \models \delta\{x := k, k\}$. We can then apply the IH to see, in particular, $\delta_1(t)\{x := k\} \sim_{\succ} \delta_2(s)\{x := k\} : \mathsf{Bool}$.

T_FunC: $\phi(\Gamma \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2) = x{:}\phi(\Gamma \vdash_c c_1 : T_1) \to \phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)$. We must show that

$$\delta_1(c_1) \sim_{\succ} \delta_2(\phi(\Gamma \vdash_c c_1 : T_1)) : T_1$$
and
$$\forall t \sim_{\succ} s : T_1.\ \delta_1(c_2)\{x := t\} \sim_{\succ} \delta_2(\phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2))\{x := s\} : T_2$$

We can see the first part by the IH on $\phi(\Gamma \vdash_c c_1 : T_1)$.

Let $t \sim_{\succ} s : T_1$. We can see $\Gamma, x{:}c_1 \models \delta\{x := t, s\} = \delta'$, so by the IH on $\phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)$, we can see

$$\delta'_1(c_2) \sim_{\succ} \delta'_2(\phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)) : T_2$$

which is exactly what we needed to show. □

We find a weak corollary: $\phi(\Gamma \vdash t : B) \longrightarrow_h^* k$ implies $t \longrightarrow_c^* k$, because if the λ_{H} term does *not* go to blame, then the original λ_{C} term must go to the same constant.

We can prove type preservation, as well, for terms not containing active checks. We don't know that translated active checks are well typed because Theorem 6.14 isn't strong enough to show that $\vdash t_2 \supset t_1\{x := k\}$ implies $\emptyset \vdash \phi(\emptyset \vdash t_2 : \mathsf{Bool}) \supset \phi(x{:}B \vdash t_1 : \mathsf{Bool})\{x := k\}$. However, since we only expect these checks to occur at runtime, this is good enough: ϕ preserves the types of source programs.

6.15 Theorem [Type preservation]: For programs without active checks, if $\phi(\vdash \Gamma) = \Delta$, then:

1. $\vdash \Delta$.
2. If $\phi(\Gamma \vdash t : T) = s$ then $\Delta \vdash s : \lceil T \rceil$.
3. If $\phi(\Gamma \vdash_c c : T) = S$ then $\Delta \vdash S$.

Proof: We prove all three properties simultaneously, by induction on the depth of ϕ's recursion.

The proof is by cases on the λ_{C} context well-formedness/term typing/contract well-formedness derivations, which determine the branch of ϕ taken.

T_Empty: $\vdash \emptyset$. By S_Empty.

T_ExtVarT: $\vdash \Gamma, x{:}T$. By the IH, $\vdash \Delta$. We have $\Delta \vdash \lceil T \rceil$ by Lemma 6.1, so $\vdash \Delta, x{:}\lceil T \rceil$ by S_ExtVar.

T_ExtVarC: $\vdash \Gamma, x{:}c$. By the IH, $\vdash \Delta$. We have $\Delta \vdash \phi(\Gamma \vdash_c c : \lfloor c \rfloor)$, so $\vdash \Delta, x{:}\phi(\Gamma \vdash_c c : \lfloor c \rfloor)$ by S_ExtVar.

T_VarT: $\phi(\Gamma \vdash x : T) = x$. By inversion, $\Gamma = \Gamma_1, x{:}T\,, \Gamma_2$. We can see $x{:}\lceil T \rceil \in \phi(\vdash \Gamma)$, so $\Delta \vdash x : \lceil T \rceil$ by S_Var.

T_VarC: $\phi(\Gamma \vdash x : \lfloor c \rfloor) = \langle \phi(\Gamma_1 \vdash_c c : T) \Rightarrow \lceil T \rceil \rangle^{l_0}\, x$, where $\Gamma = \Gamma_1, x{:}c\,, \Gamma_2$. Let $S = \phi(\Gamma_1 \vdash_c c : T)$.

By the IH, $\phi(\vdash \Gamma_1) \vdash S$; by weakening (Lemma 4.30), $\Delta \vdash S$. By Lemma 6.1, $\Delta \vdash \lceil T \rceil$. Note that $\lceil T \rceil = \lceil S \rceil$ (Lemma 6.3). By S_Cast, $\Delta \vdash \langle S \Rightarrow \lceil S \rceil \rangle^{l_0} : S \to \lceil S \rceil$. By S_Var, $\Delta \vdash x : S$. Finally, by S_App, $\Delta \vdash \langle S \Rightarrow \lceil S \rceil \rangle^{l_0}\, x : \lceil S \rceil = \lceil T \rceil$.

T_Const: $\phi(\Gamma \vdash k : \mathrm{ty_c}(k)) = k$. By S_Const, $\Delta \vdash k : \mathrm{ty_h}(k)$. By our assumption that λ_{H}'s constants' types are subtypes of their raw λ_{C} types, $\Delta \vdash \mathrm{ty_h}(k) <: \lceil \mathrm{ty_c}(k) \rceil$. Since $\Delta \vdash \mathrm{ty_h}(k)$ by assumption, we can applyS_Sub, showing $\Delta \vdash k : \lceil \mathrm{ty_c}(k) \rceil$.

Result correspondence

$$\begin{array}{c} r \approx_{\succ} q : B \iff \\ r = q = k \in \mathcal{K}_B \vee q = \Uparrow l \end{array}$$

$$\begin{array}{c} r \approx_{\succ} q : T_1 \to T_2 \iff \\ q = \Uparrow l \vee \\ (r = v \wedge q = w \wedge \\ \forall t \sim_{\succ} s : T_1.\ v\ t \sim_{\succ} w\, s : T_2) \end{array}$$

Term correspondence

$$\begin{array}{c} t \sim_{\succ} s : T \iff \\ t \longrightarrow_c^* r \wedge s \longrightarrow_h^* q \wedge r \approx_{\succ} q : T \end{array}$$

Contract / type correspondence

$$\begin{array}{c} \{x{:}B \mid t\} \sim_{\succ} \{x{:}B \mid s\} : B \iff \\ \forall k \in \mathcal{K}_B.\ t\{x := k\} \sim_{\succ} s\{x := k\} : \mathsf{Bool} \end{array}$$

$$\begin{array}{c} x{:}c_1 \mapsto c_2 \sim_{\succ} x{:}S_1 \to S_2 : T_1 \to T_2 \iff \\ c_1 \sim_{\succ} S_1 : T_1 \wedge \\ \forall t \sim_{\succ} s : T_1.\ c_2\{x := t\} \sim_{\succ} S_2\{x := s\} : T_2 \end{array}$$

Dual closing substitutions

$$\Gamma \models_{\succ} \delta \iff \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \sim_{\succ} \delta_2(x) : \lfloor \Gamma(x) \rfloor$$

Lifted to open terms

$$\begin{array}{c} \Gamma \vdash t \sim_{\succ} s : T \iff \\ \forall \delta.\ (\Gamma \models_{\succ} \delta \ \text{ implies } \ \delta_1(t) \sim_{\succ} \delta_2(s) : T) \end{array}$$

$$\begin{array}{c} \Gamma \vdash c \sim_{\succ} S : T \iff \\ \forall \delta.\ (\Gamma \models_{\succ} \delta \ \text{ implies } \ \delta_1(c) \sim_{\succ} \delta_2(S) : T) \end{array}$$

Figure 23: Blame-inexact correspondence

<u>T_Lam</u>: $\phi(\Gamma \vdash \lambda x{:}T_1.\ t_{12} : T_1 \to T_2) = \lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2)$.

By the IH on $\phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2)$, we see $\Delta, x{:}\lceil T_1 \rceil \vdash \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2) : \lceil T_2 \rceil$. By S_Lam, $\Delta \vdash \lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_{12} : T_2) : (x{:}\lceil T_1 \rceil \to \lceil T_2 \rceil)$, which is alpha-equivalent to $\lceil T_1 \to T_2 \rceil$.

<u>T_App</u>: $\phi(\Gamma \vdash t_1\, t_2 : T_2) = \phi(\Gamma \vdash t_1 : T_1 \to T_2)\, \phi(\Gamma \vdash t_2 : T_1)$.

By the IH, we see $\Delta \vdash \phi(\Gamma \vdash t_1 : T_1 \to T_2) : \lceil T_1 \to T_2 \rceil$ and $\Delta \vdash \phi(\Gamma \vdash t_2 : T_1) : \lceil T_1 \rceil$. By S_App, $\Delta \vdash \phi(\Gamma \vdash t_1\, t_2 : T_2) : \lceil T_2 \rceil\{x := \phi(\Gamma \vdash t_2 : T_1)\} = \lceil T_2 \rceil$.

<u>T_Blame</u>: $\phi(\Gamma \vdash \Uparrow l : T) = \Uparrow l$. We find $\Delta \vdash \lceil T \rceil$ by Lemma 6.1, so $\Delta \vdash \Uparrow l : \lceil T \rceil$ by S_Blame.

<u>T_Contract</u>: $\phi(\Gamma \vdash \langle c \rangle^{l,l'} : T \to T) = \lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'}\, (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x)$, where $S = \phi(\Gamma \vdash_c c : T)$. Note that $\Delta, x{:}\lceil S \rceil \vdash \lceil T \rceil$ by Lemma 6.1, that $\lceil T \rceil = \lceil S \rceil$ by Lemma 6.3, and that $\lfloor \lceil S \rceil \rfloor = \lfloor S \rfloor$ by Lemma 6.2.

By the IH for proposition (2), $\Delta \vdash S$. By weakening (Lemma 4.30), $\Delta, x{:}\lceil c \rceil \vdash S$. We can see $\Delta, x{:}\lceil S \rceil \vdash \langle S \Rightarrow \lceil S \rceil \rangle^{l'} : S \to \lceil S \rceil$ and $\Delta, x{:}\lceil S \rceil \vdash \langle \lceil S \rceil \Rightarrow S \rangle^{l} : \lceil S \rceil \to S$ by S_Cast. By S_App and S_Var, $\Delta, x{:}\lceil S \rceil \vdash \langle S \Rightarrow \lceil S \rceil \rangle^{l'}\, (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x) : \lceil S \rceil = \lceil T \rceil$. Finally, by S_Lam,

$$\Delta \vdash \lambda x{:}\lceil S \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'}\, (\langle \lceil S \rceil \Rightarrow S \rangle^{l}\, x) : (x{:}\lceil S \rceil \to \lceil S \rceil)$$

which is alpha-equivalent to $\lceil T \to T \rceil$.

<u>T_Checking</u>: Contradictory—this case is specifically excluded.

T_RefineC: $\phi(\Gamma \vdash_c \{x{:}B \mid t\} : B) = \{x{:}B \mid \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})\}$. By the IH for proposition (1) on $\phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})$, we have $\Delta, x{:}\lceil B \rceil \vdash \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool}) : \lceil \mathsf{Bool} \rceil$. By WF_Refine, $\Delta \vdash \{x{:}B \mid s\}$.

T_FunC: $\phi(\Gamma \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2) = x{:}\phi(\Gamma \vdash_c c_1 : T_1) \to \phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)$.

By the IH, $\Delta \vdash \phi(\Gamma \vdash_c c_1 : T_1)$. Note that $\phi(\vdash \Gamma, x{:}c_1) = \Delta, x{:}\phi(\Gamma \vdash_c c_1 : T_1)$. By the IH, $\Delta, x{:}\phi(\Gamma \vdash_c c_1 : T_1) \vdash \phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)$. By WF_Fun, $\Delta \vdash x{:}\phi(\Gamma \vdash_c c_1 : T_1) \to \phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)$, i.e., $\Delta \vdash \phi(\Gamma \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2)$. □

To see that the ϕ we define in Figure 22 does not give us exact blame, let us look at two counterexamples; in both cases, a λ_{C} term goes to a value while its translation goes to blame. In the first counterexample, blame is raised in λ_{H} due to F_CDecomp. In the second, blame is raised due to the extra cast from the translation of T_VarC.

First, let

$$\begin{array}{rcl} c & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\, y\}) \mapsto \\ & & \{z{:}\mathsf{Int} \mid f\,0\ =\ 0\} \\ S_1 & = & x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\, y\} \\ S & = & \phi(\emptyset \vdash_c c : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \\ & = & f{:}S_1 \to \{z{:}\mathsf{Int} \mid (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l_0} f)\, 0\ =\ 0\}. \end{array}$$

We find $\langle c \rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_c^* 0$ but $(\lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^l\,(\langle \lceil S \rceil \Rightarrow S \rangle^l\, x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l$. This is due to the mismatch between E_CDecomp and F_CDecomp.

Second, let

$$\begin{array}{rcl} c' & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \\ & & \{z{:}\mathsf{Int} \mid f\,0\ =\ 0\} \\ S_1' & = & x{:}\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\} \\ S' & = & \phi(\emptyset \vdash_c c' : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \\ & = & f{:}S_1' \to \{z{:}\mathsf{Int} \mid (\langle S_1' \Rightarrow \lceil S_1' \rceil \rangle^{l_0} f)\, 0\ =\ 0\}. \end{array}$$

We find $\langle c' \rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_c^* 0$ but $(\lambda x{:}\lceil c' \rceil.\ \langle S' \Rightarrow \lceil c' \rceil \rangle^l\,(\langle \lceil S \rceil \Rightarrow \lceil c' \rceil \rangle^l\, x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l_0$. This is due to the extra cast inserted for the sake of type preservation.

The extra casts that ϕ inserts are all necessary—none can be removed. So while variations on this ϕ are possible, they can only add more casts, which won't resolve the problem that λ_{H} blames *more*.

The nondependent restriction of ϕ

While ϕ doesn't induce an exact behavioral correspondence in the dependent case, it *does* for nondependent λ_{C} and λ_{H}, as Gronski and Flanagan [2007] have also showed. The discrepancy with regard to blame in the full system comes from two sources: the asymmetric F_CDecomp rule inserts a cast that E_CDecomp doesn't, and the translation of T_VarC derivations inserts extra casts. Neither of these come up in the nondependent case: F_CDecomp does *no* substitution when there's no dependency, and $x{:}c$ bindings only occur when we have dependent function contracts.

We give a new proof of this fact, different from Gronski and Flanagan's. It follows the same pattern as the dependent case, though of course without the extra cast lemma (6.10). The most significant change is to the correspondence relation—we use the blame-exact correspondence $\sim$ from Figure 21 in Section 5. Since we are able to prove exact behavioral correspondence, we can also show exact type preservation for the full language including active checks.

6.16 Lemma [Well-formedness of translated types]: $\vdash \lceil T \rceil$ for all T, and likewise for c and S.

Proof: By straightforward induction on T (or c or S), using WF_Raw in the base case. □

In Figure 26, the we define a behavioral correspondence between nondependent relation between λ_{C} and λ_{H}. The term correspondence is blame-exact, like that of Figure 21 in Section 5, while contract/type correspondence is similar to that used for dependent ϕ in Figure 23.

Types

$$\begin{array}{rcl}
\phi(\{x{:}B \mid t\}) & = & \{x{:}B \mid \phi(t)\} \\
\phi(c_1 \mapsto c_2) & = & \phi(c_1) \rightarrow \phi(c_2)
\end{array}$$

Terms

$$\begin{array}{rcl}
\phi(\langle c\rangle^{l,l'}) & = & \lambda x{:}\lceil c\rceil.\ \langle \phi(c) \Rightarrow \lceil c\rceil\rangle^{l'}\ (\langle \lceil c\rceil \Rightarrow \phi(c)\rangle^{l}\ x) \\
\phi(x) & = & x \\
\phi(k) & = & k \\
\phi(\lambda x{:}T.\ t) & = & \lambda x{:}\lceil T\rceil.\ \phi(t) \\
\phi(t_1\, t_2) & = & \phi(t_1)\,\phi(t_2) \\
\phi(\Uparrow l) & = & \Uparrow l \\
\phi(\langle c, t, k\rangle^{l}) & = & \langle \phi(c), \phi(t), k\rangle^{l}
\end{array}$$

Figure 24: The translation $\phi : \lambda_{\mathrm{C}} \rightarrow \lambda_{\mathrm{H}}$

Figure 25: Logical relation between λ_{C} and λ_{H} terms

Term correspondence

$\boxed{r \approx q : T}$

$$\begin{gathered}
r \approx q : B \iff \\
r = q = \Uparrow l \vee r = q = k \in \mathcal{K}_B \\
r \approx q : T_1 \rightarrow T_2 \iff \\
r = q = \Uparrow l \vee \forall t \sim s : T_1.\ r\,t \sim q\,s : T_2
\end{gathered}$$

$\boxed{t \sim s : T}$

$$\begin{gathered}
t \sim s : T \iff \\
t \longrightarrow_c^* r \wedge s \longrightarrow_h^* q \wedge r \approx q : T
\end{gathered}$$

Correspondence of open terms

$$\begin{gathered}
\Gamma \models \delta \iff \\
\forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \sim \delta_2(x) : \Gamma(x)
\end{gathered}$$

$$\begin{gathered}
\Gamma \vdash t \sim s : T \iff \\
\forall \delta.\ \Gamma \models \delta \text{ implies } \delta_1(t) \sim \delta_2(s) : T
\end{gathered}$$

Contract/type correspondence $\boxed{c \sim S : T}$

$$\begin{gathered}
\{x{:}B \mid t\} \sim \{x{:}B \mid s\} : B \iff \\
\forall k \in \mathcal{K}_B.\ t\{x := k\} \sim s\{x := k\} : \mathsf{Bool}
\end{gathered}$$

$$\begin{gathered}
c_1 \mapsto c_2 \sim S_1 \rightarrow S_2 : T_1 \rightarrow T_2 \iff \\
c_1 \sim S_1 : T_1 \wedge c_2 \sim S_2 : T_2
\end{gathered}$$

Figure 26: Nondependent correspondence between λ_{C} and λ_{H}

6.17 Lemma [Expansion and contraction]: If $t \longrightarrow_c^* t'$ and $s \longrightarrow_h^* s'$, then $t \sim s : T$ iff $t' \sim s' : T$.

Proof: By determinacy for $\longrightarrow_c^*$ (a restriction of Lemma 4.2) and $\longrightarrow_h^*$ (a restriction of Lemma 4.7). $\square$

Note that there are corresponding terms at every type.

6.18 Lemma [Blame corresponds to blame]: For all T, $\Uparrow l \sim \Uparrow l : T$.

Proof: By inspection of the definition of $\approx$ and the reflexivity of the multi-step evaluation relations. $\square$

6.19 Lemma [Constants correspond to themselves]: For all k, $k \approx k : \mathrm{ty_c}(k)$.

Proof: By induction on $\mathrm{ty_c}(k)$.

$\underline{\mathrm{ty_c}(k) = B}$: By definition.

$\underline{\mathrm{ty_c}(k) = B_1 \to B_2 \to \ldots \to B_n}$: We wish to show that $k \approx k : B_1 \to B_2 \to \ldots \to B_n$. Let $t \sim s : B_1$. By definition $t \longrightarrow_c^* r$ and $s \longrightarrow_h^* q$ such that $r \approx q : B_1$. By expansion (Lemma 5.1), it suffices to show $k\, r \sim k\, q : B_2 \to \ldots \to B_n$.

If $r = q = \Uparrow l$, then $k \Uparrow l$ reduces to $\Uparrow l$ on both sides. By Lemma 5.2, $\Uparrow l \sim \Uparrow l : B_2 \to \ldots \to B_n$; the applications are related by expansion (Lemma 5.1).

Otherwise, we have $r = q = k' \in \mathcal{K}_{B_1}$ because constants are the only values related at B_1. Since denotations are total, $[\![k]\!](k')$ is defined. Moreover, $[\![k]\!](k')$ is either a constant k'' or $\Uparrow l$, by definition. In the latter case the applications are related because they go to blame. In the former, $\mathrm{ty_c}(k'') = B_2 \to \ldots \to B_n$, so $k'' \approx k'' : B_2 \to \ldots \to B_n$ by the IH. $\square$

6.20 Lemma [Corresponding terms coevaluate]: If $t \sim s : T$ then $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ or else $t \longrightarrow_c^* \Uparrow l$ and $s \longrightarrow_h^* \Uparrow l$. Moreover, if $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$, then $v \sim w : T$.

Proof: The terms evaluate to corresponding results by definition. We can see by the definition of the relation on results that $r = \Uparrow l$ iff $q = \Uparrow l$, and likewise for values. $\square$

The previous three lemmas allow us to reason about values instead of results: corresponding terms either go to blame or values, and blame is lifted via the E_BLAME and F_BLAME rules in both λ_C and λ_H. Corresponding terms are effectively corresponding values, so an invocation of Lemma 6.20 followed by Lemma 6.18 and Lemma 6.17 will be a very common pattern in the coming proofs.

Corresponding refinements only need to correspond when we substitute approriate constants into the predicate. We do not bother to require that $t\{x := \Uparrow l\} \sim s\{x := \Uparrow l\} : \mathsf{Bool}$, since blame will never be substituted into an active check in either language. instead being lifted up by E_BLAME and F_BLAME.

6.21 Lemma [Trivial casts]: If $t \sim s : B$ then $t \sim \langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^l\ s : B$ for any s_1.

Proof: Since $t \sim s : B$, by expansion (6.17) we obtain $t \longrightarrow_c^* r$ and $s \longrightarrow_h^* q$, with $r \approx q : B$. There are two cases to consider.

- If $r = q = \Uparrow l'$, then $\langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^l\ s \longrightarrow_h^* \Uparrow l'$. Since they go to the same blame label, $t \sim \langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^l\ s : B$.
- If $r = q = k$, then $\langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^l\ k \longrightarrow_h \langle\{x{:}B \mid \mathsf{true}\}, \mathsf{true}, k\rangle^l \longrightarrow_h k$. Since $k \approx k : B$, this means $t \sim \langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^l\ s : B$. $\square$

6.22 Lemma: For all B, if

$t \sim s : B$ and
$\{x{:}B \mid t_1\} \sim \{x{:}B \mid s_1\} : B$,

then $\langle\{x{:}B \mid t_1\}\rangle^{l,l'}\ t \ \sim\ \langle\{x{:}B \mid s_2\} \Rightarrow \{x{:}B \mid s_1\}\rangle^{l}\ s\ :\ B$.

Proof: We know that t and s coevaluate either to a constant $k \in \mathcal{K}_B$ or to $\Uparrow l$. In the latter case, applying the cast and the contract will still result in blame, and we are done by Lemma 6.18 and expansion (Lemma 6.17). So suppose $t \longrightarrow_c^* k$ and $s \longrightarrow_h^* k$ where $k \in \mathcal{K}_B$. We have

$$\begin{aligned}\langle\{x{:}B \mid t_1\}\rangle^{l,l'}\ t &\longrightarrow_c^* \langle\{x{:}B \mid t_1\}, t_1\{x := k\}, k\rangle^{l}\\ \langle\{x{:}B \mid s_2\} \Rightarrow \{x{:}B \mid s_1\}\rangle^{l}\ k &\longrightarrow_h^* \langle\{x{:}B \mid s_1\}, s_1\{x := k\}, k\rangle^{l}\end{aligned}$$

Since $\{x{:}B \mid t_1\} \sim \{x{:}B \mid s_1\} : B$, we know by definition that $t_1\{x := k\} \sim s_1\{x := k\} : \mathsf{Bool}$—and so $t_1\{x := k\} \longrightarrow_c^* r$ and $s_1\{x := k\} \longrightarrow_h^* q$ with $r \approx q : \mathsf{Bool}$. These are either blame or values; the only values of type Bool are true and false, so there are three cases:

- $r = q = \Uparrow l''$. Both checks will evaluate to $\Uparrow l''$, and so the applied contract and cast correspond by Lemma 6.18.
- $r = q = \mathsf{false}$. Both checks evaluate to $\Uparrow l$, and so the applied contract and cast correspond by Lemma 6.18.
- $r = q = \mathsf{true}$. Both checks evaluate to k. Since $k \in \mathcal{K}_B$, we know $k \approx k : B$. This means the applied contract and cast both evaluate to corresponding values, so they correspond, too. □

6.23 Lemma [Bulletproofing]: For all T, if

$t \sim s : T$ and
$c \sim S : T$,

then $\langle c\rangle^{l,l'}\ t \sim \langle S \Rightarrow \lceil S\rceil\rangle^{l'}\ (\langle\lceil S\rceil \Rightarrow S\rangle^{l}\ s) : T$.

Proof: By induction on T, using Lemma 6.22 when $T = B$ or $T = B \to T_2$.

Both cases proceed by expansion (Lemma 6.17), using Lemma 6.22 and Lemma 6.21 to unwind casts at base types.

- $T = B$, so $c = \{x{:}B \mid t_1\}$ and $S = \{x{:}B \mid s_1\}$. By Lemma 6.22, we know that $\langle c\rangle^{l,l'}\ t \sim \langle\lceil S\rceil \Rightarrow S\rangle^{l}\ s : B$. Then by Lemma 6.21 we know that $\langle c\rangle^{l,l'}\ t \sim \langle S \Rightarrow \lceil S\rceil\rangle^{l'}\ \langle\lceil S\rceil \Rightarrow S\rangle^{l}\ s : B$.
- $T = T_1 \to T_2$, so $c = c_1 \mapsto c_2$ and $S = S_1 \to S_2$; also $c_1 \mapsto c_2 \sim S_1 \to S_2 : T_1 \to T_2$. We wish to show that

  $$\langle c_1 \mapsto c_2\rangle^{l,l'}\ t \sim \langle S_1 \to S_2 \Rightarrow \lceil S_1 \to S_2\rceil\rangle^{l'}\ (\langle\lceil S_1 \to S_2\rceil \Rightarrow S_1 \to S_2\rangle^{l}\ s) : T_1 \to T_2$$

  We may assume t and s go to values v and w (otherwise we are done—everything goes to blame).

  Now, suppose $t' \sim s' : T_1$. By definition, they coevaluate to corresponding results. If they go to $\Uparrow l''$, we're done: the contract and cast terms go to $\Uparrow l''$ together. So suppose $t' \longrightarrow_c^* v'$ and $s' \longrightarrow_h^* w'$. Reduction decomposes the contracts and casts:

  $$\begin{aligned}&(\langle c_1 \mapsto c_2\rangle^{l,l'}\ t)\ t' \longrightarrow_c^* \langle c_2\rangle^{l,l'}\ (v\ (\langle c_1\rangle^{l',l}\ v'))\\ &(\langle S_1 \to S_2 \Rightarrow \lceil S_1 \to S_2\rceil\rangle^{l'}\ (\langle\lceil S_1 \to S_2\rceil \Rightarrow S_1 \to S_2\rangle^{l}\ s))\ s' \longrightarrow_h^*\\ &\langle S_2 \Rightarrow \lceil S_2\rceil\rangle^{l'}\ ((\langle\lceil S_1 \to S_2\rceil \Rightarrow S_1 \to S_2\rangle^{l}\ w)\ (\langle\lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w'))\end{aligned}$$

  At this point, it isn't immediately clear which part of the term is the next redex in λ_{H}—do we turn the cast on the argument into an active check, or do we decompose the positive cast on the function? The answer depends on the form of T_1...

– If $T_1 = B$, then $c_1 = \{x{:}B \mid t_1\}$ and $S_1 = \{x{:}B \mid s_1\}$. It must be that $v' = w' = k$ for some $k \in \mathcal{K}_B$—those are the only values corresponding at B. We also know that $\{x{:}B \mid t_1\} \sim \{x{:}B \mid s_1\} : B$. By Lemma 6.22 we can see that the domain contract corresponds to the (lone) domain cast which came from decomposing the (negative) function cast:

$$\langle c_1\rangle^{l',l}\, k \sim \langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k : B$$

If these terms go to blame, we're done by Lemma 6.18; if not, they both go to k. Reducing further on the λ_{H} side, we find:

$$\begin{array}{c}(\langle S_1 \to S_2 \Rightarrow \lceil S_1 \to S_2\rceil\rangle^{l'}\ (\langle \lceil S_1 \to S_2\rceil \Rightarrow S_1 \to S_2\rangle^{l}\ s))\, s' \longrightarrow_h^* \\ \langle S_2 \Rightarrow \lceil S_2\rceil\rangle^{l'}\ (\langle \lceil S_2\rceil \Rightarrow S_2\rangle^{l}\ (w\,(\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^{l}\ k)))\end{array}$$

and then, by Lemma 6.21 we can see that the cast into raw type (which came from decomposing the positive function cast) doesn't matter

$$k \sim \langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid \mathsf{true}\}\rangle^{l}\ k : B$$

(and both *must* go to k), leaving us with:

$$\begin{array}{c}(\langle S_1 \to S_2 \Rightarrow \lceil S_1 \to S_2\rceil\rangle^{l'}\ (\langle \lceil S_1 \to S_2\rceil \Rightarrow S_1 \to S_2\rangle^{l}\ s))\, s' \longrightarrow_h^* \\ \langle S_2 \Rightarrow \lceil S_2\rceil\rangle^{l'}\ (\langle \lceil S_2\rceil \Rightarrow S_2\rangle^{l}\ (w\, k))\end{array}$$

Since $v \sim w : B \to T_2$ and $k \sim k : B$, we get $t\, k \sim s\, k : T_2$ by definition. By the IH on T_2, we obtain $\langle c_2\rangle^{l,l'}\,(v\, k) \sim \langle S_2 \Rightarrow \lceil S_2\rceil\rangle^{l'}\ (\langle \lceil S_2\rceil \Rightarrow S_2\rangle^{l}\ (w\, k)) : T_2$, completing this case.

– $c_1 = c_{11} \mapsto c_{12}$ and so $S_1 = S_{11} \to S_{12}$. Since $\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w'$ is a value, the inner function cast decomposes via F_CDecomp:

$$\begin{array}{c}\langle S_2 \Rightarrow \lceil S_2\rceil\rangle^{l'}\ (\langle \lceil S_1 \to S_2\rceil \Rightarrow S_1 \to S_2\rangle^{l}\ (w\,(\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w'))) \quad \longrightarrow_h \\ \langle S_2 \Rightarrow \lceil S_2\rceil\rangle^{l'}\ (\langle \lceil S_2\rceil \Rightarrow S_2\rangle^{l}\ (w\,(\langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ (\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w'))))\end{array}$$

By the IH for T_1, we know that $\langle c_1\rangle^{l',l}\, v' \sim \langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ (\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w') : T_1$. We have already assumed that $v \sim w : T_1 \to T_2$, so $v\,(\langle c_1\rangle^{l,l}\, v') \sim w\,(\langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ \langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w') : T_2$. By the IH for T_2 we see that the two terms correspond with the codomain contract/casts as well, so we are done by expansion (Lemma 6.17). □

The bulletproofing lemma allows us to show that t and $\phi(t)$ correspond (and, simultaneously, that c and $\phi(c)$ correspond).

6.24 Theorem [Behavioral correspondence]:

1. If $\Gamma \vdash t : T$ then $\Gamma \vdash t \sim \phi(t) : T$.
2. If $\vdash_c c : T$ then $c \sim \phi(c) : T$.

Proof: By induction on the depth of ϕ's recursion.

Let $\Gamma \models \delta$. We prove that $\delta_1(t) \sim \delta_2(\phi(t)) : T$ and $c \sim S : T$. The only interesting cases are T_Lam, which is proved by expansion (Lemma 6.17), and T_Contract, which is proved by Lemma 6.23.

Part (1) of the proof is by case analysis on the typing derivation, using the first part of the IH in all cases except for T_Contract, which uses the second part.

T_Var: $\Gamma \vdash x : T$ and $\phi(x) = x$. By $\Gamma \models \delta$ we have $\delta_1(x) \sim \delta_2(x) : T$.

T_Const: $\Gamma \vdash k : \mathrm{ty_c}(k)$ and $\phi(k) = k$. By Lemma 6.19, $k \sim k : T$.

T_Lam: $\Gamma \vdash \lambda x{:}T_1.\ t_{12} : T_1 \to T_2$ and $\phi(\lambda x{:}T_1.\ t_{12}) = \lambda x{:}\lceil T_1 \rceil.\ \phi(t_{12})$. We show the δ-closures of the two lambdas are $\approx$-related at $T_1 \to T_2$. (They're already values, so $\approx$-relation implies $\sim$-relation.)

Let $t' \sim s' : T_1$; we will show that $\delta_1(\lambda x{:}T_1.\ t_{12})\, t' \sim \delta_2(\lambda x{:}\lceil T_1 \rceil.\ \phi(t_{12}))\, s' : T_2$. First, if t' and s' go to blame, we're done, so let them go to values $v' \approx w' : T_1$. Now the applications take a step to $\delta_1(t_{12})\{x := v'\}$ and $\delta_2(\phi(t_{12}))\{x := w'\}$, respectively. By the IH on $\Gamma, x{:}T_1 \vdash t_{12} : T_2$, we can see—noting that $\Gamma, x{:}T_1 \models \delta\{x := v', w'\}$—that these two substituted terms correspond. This means that the two applications are correspond by expansion (Lemma 6.17), so the lambdas correspond.

T_App: $\Gamma \vdash t_1\, t_2 : T_2$ and $\phi(t_1\, t_2) = \phi(t_1)\, \phi(t_2)$.

By the IH we know that $\delta_1(t_1) \sim \delta_2(\phi(t_1)) : T_1 \to T_2$ and $\Gamma \vdash \delta_1(t_2) \sim \delta_2(\phi(t_2)) : T_1$. The former implies that $\delta_1(t_1) \longrightarrow_c^* v_1$ and $\delta_2(\phi(t_1)) \longrightarrow_h^* w_1$ and that $v_1\, \delta_1(t_2) \sim w_1\, \delta_2(\phi(t_2)) : T_2$. By expansion (Lemma 6.17) we can see that applications themselves correspond.

T_Blame: $\Gamma \vdash \Uparrow l : T$ and $\phi(\Uparrow l) = \Uparrow l$. We can see $\delta_1(\Uparrow l) \sim \delta_2(\Uparrow l) : T$ by Lemma 6.18.

T_Contract: $\Gamma \vdash \langle c \rangle^{l,l'} : T \to T$ and $\phi(\langle c \rangle^{l,l'}) = \lambda x{:}\lceil c \rceil.\ \langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'}\ (\langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l}\ x)$. We wish to see that the contract and its translation are correspond at $T \to T$.

By the second part of the IH, we have $c \sim \phi(c) : T$.

Let $t \sim s : T$; we prove that

$$\delta_1(\langle c \rangle^{l,l'})\, t \sim \delta_2(\lambda x{:}\lceil c \rceil.\ \langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'}\ (\langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l}\ x))\, s : T$$

If t and s go to blame, so do the contract and cast. So let $t \longrightarrow_c^* v$ and $s \longrightarrow_h^* w$ such that $v \approx w : T$ (and so $v \sim w : T$, too). By Lemma 6.23, we can see that

$$\delta_1(\langle c \rangle^{l,l'})\, v \sim \delta_2(\langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'}\ (\langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l}\ w)) : T,$$

and we are done by expansion (Lemma 6.17).

T_Checking: $\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B$ and $\phi(\langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l}) = \langle \{x{:}B \mid \phi(t_1)\}, \phi(t_2), k \rangle^{l}$, where $\emptyset \vdash k : B$ so $k \in \mathcal{K}_B$.

Noting the empty context, the IH shows $t_2 \sim \phi(t_2) : \mathsf{Bool}$. We proceed by case analysis on how $t_2 \longrightarrow_c^* r_2$ and $\phi(t_2) \longrightarrow_h^* q_2$—there are three cases:

- $r_2 = q_2 = \Uparrow l''$. Both checks step to $\Uparrow l''$.
- $r_2 = q_2 = \mathsf{false}$. Both checks step to $\Uparrow l$.
- $r_2 = q_2 = \mathsf{true}$. Both checks step to k, and $k \approx k : B$ since $k \in \mathcal{K}_B$.

In any case, the checks step to the same blame label or $\approx$-related values.

Part (2) of the proof is by case analysis on the contract well-formedness derivation.

T_BaseC: $\vdash_c \{x{:}B \mid t\} : B$ and $\phi(\{x{:}B \mid t\}) = \{x{:}B \mid \phi(t)\}$. We show that $\{x{:}B \mid t\} \sim \{x{:}B \mid \phi(t)\} : B$.

By the first part of the IH on $x{:}B \vdash t : \mathsf{Bool}$, we know that $x{:}B \vdash t \sim \phi(t) : \mathsf{Bool}$. This means that, in particular, $t\{x := k\} \sim \phi(t)\{x := k\} : \mathsf{Bool}$—and so the contract and cast are related.

T_FunC: $\vdash_c c_1 \mapsto c_2 : T_1 \to T_2$ and $\phi(c_1 \mapsto c_2) = \phi(c_1) \to \phi(c_2)$. We show that $c_1 \mapsto c_2 \sim \phi(c_1) \to \phi(c_2) : T_1 \to T_2$.

By the second part of the IH, we find $c_1 \sim \phi(c_1) : T_1$ and $c_2 \sim \phi(c_2) : T_2$, which is enough to see $c_1 \mapsto c_2 \sim \phi(c_1) \to \phi(c_2) : T_1 \to T_2$. ☐

The correspondence theorem gives a strong corollary: t and $\phi(t)$ yield syntactically equal results at base types.

6.25 Corollary: If $\Gamma \vdash t : T$ then

1. $t \longrightarrow_c^* \Uparrow l$ iff $s \longrightarrow_h^* \Uparrow l$, and
2. if $T = B$, then $t \longrightarrow_c^* k$ iff $\phi(t) \longrightarrow_h^* k$.

We prove that ϕ preserves the implication judgment.

6.26 Lemma: If

$\emptyset \vdash t_1 : \mathsf{Bool}$,
$\emptyset \vdash t_2 : \mathsf{Bool}$, and
$\vdash t_1 \supset t_2$,

then $\vdash \phi(t_1) \supset \phi(t_2)$.

Proof: By the logical relation. Lemma 6.24 gives us $t_1 \sim \phi(t_1) : \mathsf{Bool}$ and $t_2 \sim \phi(t_2) : \mathsf{Bool}$.

Suppose $\phi(t_1) \longrightarrow_h^* \mathsf{true}$. Then we must have $t_1 \longrightarrow_c^* \mathsf{true}$, so $t_2 \longrightarrow_c^* \mathsf{true}$, too. This means that $\phi(t_2) \longrightarrow_h^* \mathsf{true}$. □

Using the behavioral correspondence theorem, we show that ϕ preserves typing (Theorem 6.27). Note that we *must* do the proof in this order, to handle the typing rule for checks.

6.27 Theorem [Type preservation]:

1. If $\Gamma \vdash t : T$ then $\lceil\Gamma\rceil \vdash \phi(t) : \lceil T\rceil$.
2. If $\vdash_c c : T$ then $\vdash \phi(c)$.

Proof: We simultaneously show both properties by induction on the depth of ϕ's recursion.

Part (1) of the proof proceeds by case analysis on the final rule of t's typing derivation.

T_VAR: $\Gamma \vdash x : T$ and $\phi(x) = x$. We must have $x{:}T \in \Gamma$, so $x{:}\lceil T\rceil \in \lceil\Gamma\rceil$. By S_VAR, $\lceil\Gamma\rceil \vdash x : \lceil T\rceil$¿

T_CONST: $\Gamma \vdash k : \mathrm{ty_c}(k)$ and $\phi(k) = k$. We can see $\lceil\Gamma\rceil \vdash \phi(k) : \mathrm{ty_h}(k)$ by S_CONST. Note that $\lceil \mathrm{ty_c}(k)\rceil$ is well-formed by Lemma 6.16.

If $\mathrm{ty_c}(k) = B$, then we know that $\vdash \mathrm{ty_h}(k) <: \lceil \mathrm{ty_c}(k)\rceil$ since constants have most-specific types and the type assignment functions agree on skeletons. $\lceil\Gamma\rceil \vdash k : \lceil \mathrm{ty_c}(k)\rceil$ by S_SUB.

If $\mathrm{ty_c}(k) = T_1 \to T_2$, then we have assumed that $\vdash \mathrm{ty_h}(k) <: \lceil \mathrm{ty_c}(k)\rceil$, so by S_SUB.

T_LAM: $\Gamma \vdash \lambda x{:}T_1.\ t_{12} : T_1 \to T_2$ and $\phi(\lambda x{:}T_1.\ t_{12}) = \lambda x{:}\lceil T_1\rceil.\ \phi(t_{12})$.

By the IH, we know that $\lceil\Gamma\rceil, x{:}\lceil T_1\rceil \vdash \phi(t_{12}) : \lceil T_2\rceil$. We have $\vdash \lceil T_1\rceil$ by Lemma 6.16, so $\lceil\Gamma\rceil \vdash \lambda x{:}\lceil T_1\rceil.\ \phi(t_{12}) : \lceil T_1 \to T_2\rceil$ by S_LAM.

T_APP: $\Gamma \vdash t_1\, t_2 : T_2$ and $\phi(t_1\, t_2) = \phi(t_1)\, \phi(t_2)$.

By the IH we have $\lceil\Gamma\rceil \vdash \phi(t_1) : \lceil T_1 \to T_2\rceil$ and $\lceil\Gamma\rceil \vdash \phi(t_2) : \lceil T_1\rceil$. We can combine the two with T_APP to find $\lceil\Gamma\rceil \vdash \phi(t_1\, t_2) : \lceil T_2\rceil$.

T_BLAME: $\Gamma \vdash \Uparrow l : T$ and $\phi(\Uparrow l) = \Uparrow l$.

We see $\vdash \lceil T\rceil$ by Lemma 6.16. $\lceil\Gamma\rceil \vdash \Uparrow l : \lceil T\rceil$ by S_BLAME.

T_Contract: $\Gamma \vdash \langle c \rangle^{l,l'} : T \to T$ and $\phi(\langle c \rangle^{l,l'}) = \lambda x{:}\lceil c \rceil.\ \langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'} \ (\langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l} \ x)$. We show that latter is well-typed.

By the IH on $\vdash_c c : T$, $\vdash \phi(c)$. Note that $\lfloor \phi(c) \rfloor = \lfloor \lceil c \rceil \rfloor$. This is enough to see that $\lceil \Gamma \rceil, x{:}\lceil c \rceil \vdash \langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'} : \phi(c) \to \lceil c \rceil$ and $\lceil \Gamma \rceil, x{:}\lceil c \rceil \vdash \langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l} : \lceil c \rceil \to \phi(c)$ by S_Cast. By S_Var we have $\lceil \Gamma \rceil, x{:}\lceil c \rceil \vdash x : \lceil c \rceil$, so the application yields $\lceil \Gamma \rceil, x{:}\lceil c \rceil \vdash \langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'} \ (\langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l} \ x) : \lceil c \rceil$. By S_Lam, $\lceil \Gamma \rceil \vdash \lambda x{:}\lceil c \rceil.\ \langle \phi(c) \Rightarrow \lceil c \rceil \rangle^{l'} \ (\langle \lceil c \rceil \Rightarrow \phi(c) \rangle^{l} \ x) : \lceil c \rceil \to \lceil c \rceil$.

T_Checking: $\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B$ and $\phi(\langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l}) = \langle \{x{:}B \mid \phi(t_1)\}, \phi(t_2), k \rangle^{l}$. We show the latter is well-typed. We show each of the four necessary premises.

By the IH on $x{:}B \vdash t_1 : \mathsf{Bool}$ we have $x{:}\lceil B \rceil \vdash \phi(t_1) : \lceil \mathsf{Bool} \rceil$, so $\vdash \{x{:}B \mid \phi(t_1)\}$.

By the IH on $\emptyset \vdash t_2 : \mathsf{Bool}$ we have $\emptyset \vdash \phi(t_2) : \lceil \mathsf{Bool} \rceil$.

By the IH on $\emptyset \vdash k : B$ we have $\emptyset \vdash k : \lceil B \rceil$.

By Lemma 6.26, $\vdash t_2 \supset t_1\{x := k\}$ implies $\vdash \phi(t_2) \supset \phi(t_1)\{x := k\}$.

Part (2) of the proof is by case analysis on the final rule used in the contract well-formedness derivation.

T_BaseC: $\vdash_c \{x{:}B \mid t\} : B$ and $\phi(\{x{:}B \mid t\}) = \{x{:}B \mid \phi(t)\}$.

By the IH on $x{:}B \vdash t : \mathsf{Bool}$ we have $x{:}\lceil B \rceil \vdash \phi(t) : \lceil \mathsf{Bool} \rceil$, so $\vdash \{x{:}B \mid \phi(t)\}$ by WF_Refine.

T_FunC: $\vdash_c c_1 \mapsto c_2 : T_1 \to T_2$ and $\phi(c_1 \mapsto c_2) = \phi(c_1) \to \phi(c_2)$.

By the IH on $\vdash_c c_1 : T_1$ and $\vdash_c c_2 : T_2$, we find $\vdash \phi(c_1)$ and $\vdash \phi(c_2)$. By WF_Fun, $\vdash \phi(c_1) \to \phi(c_2)$. □

Having shown a strong behavioral correspondence for nondependent ϕ, we show that translating derivations gives the same output for nondepedendent inputs as the nondependent ϕ discussed in Section 3.

6.28 Lemma: If dependency in function contracts is not used in t or c, then

1. If $\Gamma \vdash t : T$, then $\phi(t) = \phi(\Gamma \vdash t : T)$, and
2. If $\Gamma \vdash_c c : T$, then $\phi(c) = \phi(\Gamma \vdash_c c : T)$.

Proof: By mutual induction on t and c.

- $t = x$. Since dependency in function contracts isn't used, we know that $x{:}T \in \Gamma$—and not $x{:}c$—so $\phi(x) = \phi(\Gamma \vdash x : T) = x$.
- $t = k$. By definition, $\phi(\Gamma \vdash k : T) = k$.
- $t = \lambda x{:}T_1.\ t_2$. By the IH, $\phi(t_2) = \phi(\Gamma, x{:}T_1 \vdash t_2 : T_2)$. The type annotation is $\lfloor T_1 \rfloor$ in both.
- $t = t_1\, t_2$. By the IH.
- $t = \Uparrow l$. By definition, $\phi(\Gamma \vdash \Uparrow l : T) = \Uparrow l$.
- $t = \langle c \rangle^{l,l'}$. We see $\phi(\Gamma \vdash \langle c \rangle^{l,l'} : T \to T) = \lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^{l'} \ (\langle \lceil S \rceil \Rightarrow S \rangle^{l} \ x)$ where $S = \phi(\Gamma \vdash_c c : T)$ and x is fresh. By the IH, $S = \phi(c)$, and both translations generate the same "plumbing" in the lambda.
- $t = \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l}$. By the IH.
- $c = \{x{:}B \mid t\}$. By the IH.
- $c = x{:}c_1 \mapsto c_2$. By the IH, $\phi(c_1) = \phi(\Gamma \vdash_c c_1 : T_1)$. We know that x isn't referenced in c_2, so $\phi(c_2) = \phi(\Gamma, x{:}c_1 \vdash_c c_2 : T_2)$ by the IH. Finally, $\phi(c_1 \mapsto c_2) = \phi(\Gamma \vdash_c x{:}c_1 \mapsto c_2 : T_1 \to T_2)$. □

	latent systems					manifest systems					
	FF02	HJL06	GF07 λ_{C}	BM06	our λ_{C}	GF07 λ_{H}	F06	KF09	WF09	OTMW04	our λ_{H}
	(1)	(2)	(3)	(4)		(3)	(5)	(6)	(7)	(8)	
dep (9)	✓ lax	✓ picky	×	(10)	✓ lax	×	✓	✓	×	✓	✓
eval order	CBV	lazy	CBV	CBV	CBV	CBV	CBN(11)	full β	CBV	CBV	CBV
blame (12)	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$ or $\perp$	$\Uparrow l$	$\Uparrow l$	stuck	stuck	$\Uparrow l$	$\Uparrow$	$\Uparrow l$
checking (13)	if	if	$\bigcirc$	active	active	$\bigcirc$	$\bigcirc$	active	active	if	active
typing (14)	✓	✓	✓	n/a	✓	×	×	✓	✓	✓	✓
any con (15)	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓

(1) Findler and Felleisen [2002]. (2) Hinze et al. [2006]. (3) Gronski and Flanagan [2007]. (4) Blume and McAllester [2006]. (5) Flanagan [2006]. (6) Knowles and Flanagan [2009]. (7) Wadler and Findler [2009]. (8) Ou et al. [2004]. (9) Does the system include dependent contracts or function types (✓) or not (×) and, for contracts, is the semantics lax or picky? (10) An "unusual" form of dependency, where negative blame in the codomain results in nontermination. (11) A nondeterministic variant of CBN. (12) Do failed contracts raise labeled blame ($\Uparrow l$), raise blame without a label ($\Uparrow$), get stuck, or sometimes raise blame and sometimes diverge ($\perp$)? (13) Is contract or cast checking performed using an "active check" syntactic form (active), an "if" construct with a refined typing rule (if), or "inlined" by making the operational semantics refer to its own reflexive and transitive closure ($\bigcirc$)? (14) Is the typing relation well defined (i.e., for dependently typed systems, is it based on a type semantics or, as in WF09, a "tagging" scheme), or is the definition circular? (15) Are arbitrary user-defined boolean functions allowed as contracts or refinements (✓), or only built-in ones (×)?

Figure 27: Comparison of related systems

7 Related work

Programming languages conferences in recent years have seen a profusion of papers on higher-order contracts and related features. This is all to the good, but, for newcomers to the area, it can be a bit overwhelming—especially given the great variety of technical approaches. To help reduce the level of confusion, in Figure 27 we summarize the important points of comparison between a number of systems that are closely related to ours.

The largest difference (though, oddly, it is rarely discussed) is between latent and manifest treatments of contracts—whether contract checking (or whatever it is called in a given system) is a completely dynamic matter or whether it leaves a "trace" that the type system can track.

Another major distinction (labeled "dep" in the figure) is the presence of dependent contracts (or dependent function types, in manifest systems). Latent systems with dependent contracts also vary in whether their semantics is lax or picky (see Section 4).

Next, most contract calculi use a standard call-by-value order of evaluation ("eval order" in the figure). Notable exceptions include Hinze et al. [2006], which is embedded in Haskell, Flanagan [2006], which uses a variant of call-by-name, and Knowles and Flanagan [2009], which uses full β-reduction (we return to this point below).

Another point of variation ("blame" in the figure) is how contract violations or cast failures are reported—by raising an exception or by getting stuck. We return to this below.

The next two rows in the table ("checking" and "typing") concern more technical points in the papers most closely related to ours. In both Gronski and Flanagan [2007] and Flanagan [2006], the operational semantics checks casts "all in one go":

$$\frac{s_2\{x := k\} \longrightarrow_h^* \mathsf{true}}{\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\ k \longrightarrow_h k}$$

Such rules are formally awkward, and in any case they violate the spirit of a small-step semantics. Also, the formal definitions of λ_{H} in both Gronski and Flanagan [2007] and Flanagan [2006] involve a circularity between the typing, subtyping, and implication relations. Knowles and Flanagan [2009] improves the technical presentation of λ_{H} in both respects. In particular, it introduces (as we do) a denotational interpretation of contract types, avoiding the definitional circularity, and it introduces a new syntactic form of "partially evaluated casts" (like most of the other systems) to maintain a small-step evaluation regime.

Our main contributions are (1) the translations ϕ and ψ and their properties, and (2) the formulation and metatheory of dependent λ_{H}. (Dependent λ_{C} is not a contribution on its own: many similar systems have been studied, and in any case its properties are much easier.) The non-dependent part of our ϕ translation essentially coincides with the one studied by Gronski and Flanagan [2007] for non-dependent λ_{C} and λ_{H}, and our behavioral correspondence theorem is essentially the same as theirs. Our ψ translation completes their story for the non-dependent case, establishing a tight connection between the systems. The full dependent forms of ϕ and ψ are novel, as is the observation that the correspondence between the latent and manifest world is more complex in this setting.

Our formulation of λ_{H} is most comparable to that of Knowles and Flanagan [2009], but there are some significant differences. First, our cast-checking constructs are equipped with labels and failed casts go to explicit blame—i.e., they raise labeled exceptions. In the λ_{H} of Knowles and Flanagan (though not the earlier one of Gronski and Flanagan), failed casts are simply stuck terms (their Progress theorem says "If a well-typed term cannot step, then either it is a value or it contains a stuck cast"). Second, their operational semantics uses full, non-deterministic β-reduction, rather than specifying a particular order of reduction, as we have done. This greatly simplifies parts of the metatheory by allowing them to avoid introducing parallel reduction. We prefer to stick with standard call-by-value reduction because we consider blame as an exception—a computational effect—and we care about *which* blame will be raised by expressions involving many casts.

The system studied by Ou et al. [2004] is also quite close in spirit to our λ_{H}. The main difference is that, because their system includes general recursion, they restrict the terms that can appear in contracts to just applications involving predefined constants: only "pure" terms can be substituted into types, and these do not include lambda-abstractions. Our system (like all of the others in Figure 27—see the row labeled "any con") allows arbitrary user-defined boolean functions to be used as contracts.

Our description of λ_{C} is ultimately based on λ_{CON} Findler and Felleisen [2002], though our presentation is slightly different in its use of checks. Hinze et al. [2006] adapted Findler and Felleisen-style to a location-passing implementation in Haskell. Notably, their dependent function contract rule is picky, not lax like ours (and Findler and Felleisen's).

Our λ_{H} type semantics in Section 4 is effectively a semantics of contracts. Blume and McAllester [2006] offers a semantics of contracts that is slightly different — our semantics includes blame at every type, while theirs explicitly excludes it. Xu et al. [2009] is also similar, though their "contracts" have no dynamic semantics at all—they are simply specifications.

We have discussed only a small sample of the many recent (and classic) papers on contracts and related ideas. We refer the reader to Knowles and Flanagan [2009] for a more comprehensive survey. Another useful resource is Wadler and Findler [2007] (technically superceded by Wadler and Findler [2009], but with a longer related work section), which surveys work combining contracts with type `Dynamic` and related features.

There are also *many* other systems that use precise types of various sorts in a completely static manner. One notable example is the work of Xu et al. [2009], which uses user-defined boolean predicates to classify values (justifying their use of the term 'contracts'), but which checks statically that these predicates hold.

It is worth mentioning that Sage [Gronski et al., 2006] and Knowles and Flanagan [2009] both support mixed static and dynamic checking of contracts, using, e.g., a theorem prover. Our work does not address that aspect, since we work with the core calculus λ_{H}.

8 Future work

Our λ_{C} and λ_{H} are strongly normalizing; extending our results to systems that allow recursion is a natural next step. The changes seem nontrivial: with the introduction of nontermination, $\sim_{\succ}$ has to allow not only more blame, but more nontermination in λ_{H}—running more casts means more opportunities for divergence.

Most studies of contracts (including ours) only allow refinements of base types; however, Blume and McAllester [2006] and Xu et al. [2009] also allow refinements of functions. This extension seems needed if contracts are to be combined with polymorphism, since in this setting we may want to refine type variables, which may later be substituted with types involving functions. We conjecture that dependent λ_{H} with function refinements is sound, but it is not clear how the translations will need to be modified.

Our operational semantics for dependent λ_{C} used the lax E_CDecomp rule. Although the picky E_CDecomp rule is also sound, we have so far been unable to come up with corresponding versions of ψ and ϕ. It seems likely that it is not possible to find translations that establish an exact behavioral correspondence (as in the non-dependent case), but one might hope for a weaker correspondence, analogous to the one we've presented here.

9 Conclusion

We can faithfully encode dependent λ_{H} into λ_{C}—the behavioral correspondence is tight. λ_{H}'s F_CDecomp rule forces us to accept a weaker behavioral correspondence when encoding λ_{C} into λ_{H}, so we conclude that the manifest and latent approaches are *not* equivalent in the dependent case. We do find, however, that the two approaches are entirely inter-encodable in the nondependent restriction.

Acknowledgements

Sewell and Zappa Nardelli's OTT tool Sewell et al. [2007] was invaluable in keeping our technicalities organized. Nate Foster joined us in many early discussions about contracts. Chris Casinghino and Nate Foster gave us helpful comments on an early draft. Chris Casinghino and Brian Aydemir helped formalize the parallel-reduction proofs.

References

Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. `http://www.cis.upenn.edu/~baydemir/papers/lngen.pdf`, March 2009.

Matthias Blume and David A. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In *Information and Computation*, pages 750–770. Springer-Verlag, 1991.

Olaf Chitil and Frank Huch. Monadic, prompt lazy assertions in haskell. In *APLAS*, pages 38–53, 2007.

Coq development team. The Coq proof assistant reference manual, version 8.2, August 2009. `http://coq.inria.fr/`.

Robert Bruce Findler. Contracts as pairs of projections. In *Symposium on Logic Programming*, pages 226–241, 2006.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, 2002.

Cormac Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.

Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, 2007.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS*, pages 29–40, 2007.

Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Functional and Logic Programming (FLOPS)*, pages 208–225, 2006.

Kenneth Knowles and Cormac Flanagan. Hybrid type checking. To appear in TOPLAS., 2009.

Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In *ICFP*, pages 1–12, 2007.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Principles of Programming Languages (POPL)*, pages 395–406, 2008.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, 2007.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, 2009.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *Principles of Programming Languages (POPL)*, pages 41–52, 2009.