Under consideration for publication in J. Functional Programming

Contracts Made Manifest

MICHAEL GREENBERG, BENJAMIN C. PIERCE, STEPHANIE WEIRICH
University of Pennsylvania

Abstract

Since Findler & Felleisen (2002) introduced *higher-order contracts*, many variants 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 to each value. These two approaches are commonly 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 & 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 from λ_{H} to λ_{C} and prove an analogous theorem.

We then generalize the whole framework to *dependent contracts*, whose predicates can mention free variables. This extension is both pragmatically crucial, supporting a much more interesting range of contracts, and theoretically challenging. We define dependent versions of λ_{H} and two dialects ("lax" and "picky") of λ_{C}, establish type soundness—a substantial result in itself, for λ_{H}—and extend ϕ and ψ accordingly. Surprisingly, the intuition that the latent and manifest systems are equivalent now breaks down: the extended translations preserve behavior in one direction but, in the other, sometimes yield terms that blame more.

This is a longer version of a POPL 2010 paper (Greenberg et al.*, 2010), with proofs and extended discussion.*

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 & 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 $\mathsf{pos}\ 1$ returns $\mathtt{true}$. On the other hand, the term $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^{l,l'}\ 0$ evaluates to *blame*, written $\Uparrow l$, 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

$$\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^{l,l'}\ (\lambda x{:}\mathsf{Int}.\ x - 1)$$

"wraps" the function $\lambda x{:}\mathsf{Int}.\ x - 1$ in a pair of checks: whenever the wrapped function is called, the argument is checked to see whether it is nonzero; 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 & 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 come in two different sorts. In systems with *latent* contracts, types and contracts are orthogonal features. Examples of this style include Findler and Felleisen's original system, Hinze *et al.* (2006), Blume & McAllester (2006), Chitil & Huch (2007), Guha *et al.* (2007), and Tobin-Hochstadt & Felleisen (2008). By contrast, *manifest* contracts are integrated into 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 & Findler (2009), and Knowles *et al.* (2006).

The key feature of manifest systems is that descriptions 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 really is nonzero, raising $\Uparrow l$ otherwise. 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.

The manifest analogue of function contracts is casts between function types. For example, consider:

$$f \;=\; \langle \lceil \mathsf{Int} \rceil \to \lceil \mathsf{Int} \rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \to \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \rangle^l\ (\lambda x{:}\lceil \mathsf{Int} \rceil.\ x - 1),$$

where $\lceil \mathsf{Int} \rceil = \{x{:}\mathsf{Int} \mid \mathsf{true}\}$. The sequence of events when f is applied to some argument n (of type P) is similar to what we saw before:

$$f\ n \longrightarrow_h \langle \lceil \mathsf{Int} \rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \rangle^l\ ((\lambda x{:}\lceil \mathsf{Int} \rceil.\ x - 1)\ (\langle \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \Rightarrow \lceil \mathsf{Int} \rceil \rangle^l\ n))$$

First, n is cast from $\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$ to $\lceil \mathsf{Int} \rceil$ (it happens that in this case the 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 its result is cast from $\lceil \mathsf{Int} \rceil$ to $\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$, raising $\Uparrow l$ if this fails. The domain cast is contravariant and the codomain cast is covariant.

One point to note here is that casts in the manifest system have just one label, while contract checks in the latent system have two. This difference is not fundamental to the latent/manifest distinction—both latent and manifest systems can be given more or less rich algebras of blame—but rather a question of the pragmatics of assigning responsibility: contract checks (called *obligations* in Findler & Felleisen (2002)) use two casts, while casts use one. Informally, a function contract check $\langle c_1 \mapsto c_2 \rangle^{l,l'}\ f$ divides responsibility for f's behavior between its body and its environment: the programmer is saying "If f is ever applied to an argument that does not pass c_1, I refuse responsibility ($\Uparrow l'$), whereas if f's result for good arguments does not satisfy c_2, I accept responsibility ($\Uparrow l$)." In a system with casts, the programmer who writes $\langle R_1 \to R_2 \Rightarrow S_1 \to S_2 \rangle^l\ f$ is saying "Although all I know statically about f is that its results satisfy R_2 when it is applied to arguments satisfying R_1, I assert that it's okay to use it on arguments satisfying S_1 [because I believe that S_1 implies R_1] and that its results will 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 (by another cast, with a different label).

While contract checks in latent systems may seem intuitively to be much the same thing as casts in manifest systems, the formal correspondence is not immediate. How do the contravariant function casts of λ_H relate to the invariant checks of λ_C? How does λ_H model λ_C's pair of polarized blame labels? This has led to some confusion in the community about the nature of contracts. Indeed, as we will see, matters become yet murkier in richer languages with features such as dependency.

Gronski & 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 a simply typed lambda-calculus, and an analogous manifest calculus λ_H. To compare these systems, they introduced a type-preserving translation ϕ from λ_C to λ_H. What makes ϕ interesting is that it relates the languages *feature for feature*: contracts over base types are mapped to casts at base type, and function contracts are mapped to function casts. The main result is that ϕ preserves behavior, in the sense that if a term t in λ_C evaluates to a constant k or blame $\Uparrow l$, then its translation $\phi(t)$ evaluates similarly.

Our work extends theirs in two directions. First, we strengthen their main result by introducing a new feature-for-feature translation ψ from λ_H to λ_C and proving a similar correspondence theorem for ψ. (We also give a new, more detailed, proof of the correspondence theorem for ϕ.) These correspondences show that the manifest and latent approaches are effectively equivalent in the nondependent case.

Second, and more significantly, we extend the whole story to allow dependent function contracts in λ_C and dependent arrow types in λ_H. Dependency is extremely handy in contracts, as it allows for precise specifications of how the results of functions depend on their arguments. For example, here is a contract that we might use with 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 easy: the dependency is all in the contracts and the types stay simple. We have just one significant design choice: should domain contracts be rechecked when the bound variable appears the codomain contract? This choice leads to two dialects of λ_C, one which does recheck (*picky* λ_C) and one which does not (*lax* λ_C). The choice is not clear, so we consider both. The question of which blame labels belong on this extra check is discussed at length in Dimoulas *et al.* (2011), which introduces *indy* blame. Indy blame is a variant of picky. We do not consider it in depth here, since it does not affect *whether or not* blame is raised, only *which* blame. We discuss this point more in Section 6.3. In λ_H, on the other hand, 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.

Surprisingly, the tight correspondence between λ_C and λ_H breaks down in the dependent case: the natural generalization of the translations does not preserve behavior exactly. Indeed, we can place λ_H between the two variants of λ_C on an "axis of blame" (Figure 1), where evaluation behavior is preserved exactly when moving left on the axis (from picky λ_C to λ_H to lax λ_C), but translated terms can blame more than their pre-images when

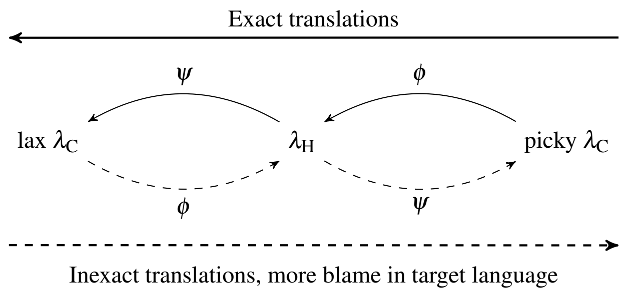


Fig. 1. The axis of blame

moving right.[1] It is still the case that when a pre-image raises blame, its translation blames as well—though not necessarily the same label. The discrepancy arises in the case of "abusive" contracts, such as

$$f{:}(\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\}$$

This rather strange contract has the form $f{:}c_1 \mapsto c_2$, where c_2 uses f in a way that violates c_1! In particular, if we apply it (in lax λ_C) 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, running the translation of f in λ_H yields an extra check, wrapping the occurrence of f in the codomain contract with a cast from $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\}$ to $\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\}$, which fails when the wrapped function is applied to 0. We discuss this phenomenon in greater detail in Section 4.

We should note at the outset that, like Gronski & Flanagan (2007), we are interested in translations that relate λ_C and λ_H feature for feature, i.e., mapping base contracts to base contracts and function contracts to function contracts. This translation, in contrast to the dependent version of the wrap operator from Findler & Felleisen (2002), works fine as a translation, giving us an exact treatment of blame. The wrap operator works on two cases: one for refinements of base types B, one for dependent function contracts.

$$\begin{array}{rcl}
\phi(\langle\{x{:}B \mid t\}\rangle^{l,l'}) & = & \langle\lceil B\rceil \Rightarrow \{x{:}B \mid \phi(t)\}\rangle^{l} \\
\phi(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}) & = & \lambda f{:}\lceil x{:}c_1 \mapsto c_2\rceil. \\
& & \quad \lambda x{:}\lceil c_1\rceil. \\
& & \quad\quad \phi(\langle c_2\rangle^{l,l'})\ (f\ (\phi(\langle c_1\rangle^{l',l})\ x))
\end{array}$$

We can define a similar mapping function that implements λ_H's semantics as base type contracts in lax or picky λ_C. This is unsurprising: λ_C and λ_H are lambda calculi that feature, among other things, a way to conditionally raise exceptions. That these languages are interencodable is completely unsurprising. But translations like these do not relate function contracts to function casts at all, so they do not do much to tell us about how semantics of contracts and the semantics of casts relate.

[1] There might, in principle, be some other way of defining ϕ and ψ that (a) preserves types, (b) maps feature for feature, and (c) induces an exact behavioral equivalence. After considering a number of alternatives, we conjecture that no such ϕ and ψ exist.

$$\begin{array}{lcll} B & ::= & \mathsf{Bool} \mid \dots & \text{base types} \\ k & ::= & \mathsf{true} \mid \mathsf{false} \mid \dots & \text{first-order constants} \end{array}$$

Fig. 2. Base types and constants for λ_C and λ_H

In summary, our main contributions are (a) the translation ψ and a symmetric version of Gronski & 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 with regard to λ_H and both dialects of λ_C, and (d) a weaker behavioral correspondence in the dependent case. We restrict ourselves to strongly normalizing programs, though we believe the results should generalize readily to programs with recursion and nontermination. This paper extends the discussion of Greenberg *et al.* (2010), giving the more interesting proofs.

2 The nondependent languages

We begin in this section by defining the nondependent versions of λ_C and λ_H and continue in Section 3 with the translations between them. The dependent languages, dependent translations, and their properties are developed in Sections 4, 5, and 6. Throughout the paper, rules prefixed with an E or an F are operational rules for λ_C and λ_H, respectively. An initial T is used for λ_C typing rules; typing rules beginning with an S belong to λ_H.

All of our languages will share a set of base types and first-order constants, given in Figure 2. Let the set $\mathscr{K}_B$ contain constants of base type B. We assume that Bool is among the base types, with $\mathscr{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

The language λ_C

The language λ_C is the simply typed lambda calculus straightforwardly augmented with contracts. *Contracts* c come in two forms: 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. We can use contracts in terms with the *contract obligation* $\langle c\rangle^{l,l'}$. Applying a contract obligation $\langle c\rangle^{l,l'}$ to a term t dynamically ensures that t and its surrounding context satisfy c. If t does not 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 does not treat $\langle c\rangle^{l,l'}\ t$ as c demands, then the negative label l' will be blamed and the term will reduce to $\Uparrow l'$. In contexts where it is unambiguous, we refer to contract obligations simply as contracts.

The syntax and semantics of λ_C appears in Figure 3, with some common definitions (shared with λ_H) in Figure 2. Besides the contract term $\langle c\rangle^{l,l'}$, λ_C 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 a technical artifact of 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 base contracts over functions themselves, like $\{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\ \mathsf{true} = f\ \mathsf{false}\}$.

Values v include constants, abstractions, contracts, and function contracts applied to values (more on these later); a *result* r is either a value or $\Uparrow l$ for some l. We interpret

Syntax for λ_C

$$
\begin{array}{rcll}
T & ::= & B \mid T_1 \to T_2 & \text{types} \\
c & ::= & \{x{:}B \mid t\} \mid c_1 \mapsto c_2 & \text{contracts} \\
\\
t & ::= & x \mid k \mid \lambda x{:}T_1.\ t_2 \mid t_1\ t_2 \mid \Uparrow l \mid \langle c \rangle^{l,l'} \mid \langle \{x{:}B \mid t_1\}, t_2, k\rangle^{l} & \text{terms} \\
v & ::= & k \mid \lambda x{:}T_1.\ t_2 \mid \langle c \rangle^{l,l'} \mid \langle c_1 \mapsto c_2 \rangle^{l,l'}\ v & \text{values} \\
r & ::= & v \mid \Uparrow l & \text{results} \\
E & ::= & [\,]\ t \mid v\ [\,] \mid \langle \{x{:}B \mid t\}, [\,], k\rangle^{l} & \text{evaluation contexts}
\end{array}
$$

Operational semantics for λ_C

$$
\begin{array}{rcll}
(\lambda x{:}T_1.\ t_2)\ v & \longrightarrow_c & t_2\{x := v\} & \text{E\_BETA} \\
k\ v & \longrightarrow_c & [\![k]\!](v) & \text{E\_CONST} \\
\langle \{x{:}B \mid t\} \rangle^{l,l'}\ k & \longrightarrow_c & \langle \{x{:}B \mid t\}, t\{x := k\}, k\rangle^{l} & \text{E\_CCHECK} \\
\langle \{x{:}B \mid t\}, \mathsf{true}, k\rangle^{l} & \longrightarrow_c & k & \text{E\_OK} \\
\langle \{x{:}B \mid t\}, \mathsf{false}, k\rangle^{l} & \longrightarrow_c & \Uparrow l & \text{E\_FAIL} \\
(\langle c_1 \mapsto c_2 \rangle^{l,l'}\ v)\ v' & \longrightarrow_c & \langle c_2 \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) & \text{E\_CDECOMP} \\
E[\Uparrow l] & \longrightarrow_c & \Uparrow l & \text{E\_BLAME} \\
E[t_1] & \longrightarrow_c & E[t_2] \quad \text{when } t_1 \longrightarrow_c t_2 & \text{E\_COMPAT}
\end{array}
$$

Typing rules for λ_C

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

$$
\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_VAR}
\qquad
\frac{}{\Gamma \vdash k : \mathrm{ty_c}(k)}\ \text{T\_CONST}
$$

$$
\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2}\ \text{T\_LAM}
\qquad
\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{T\_APP}
$$

$$
\frac{\vdash c : T}{\Gamma \vdash \langle c \rangle^{l,l'} : T \to T}\ \text{T\_CONTRACT}
\qquad
\frac{}{\Gamma \vdash \Uparrow l : T}\ \text{T\_BLAME}
$$

$$
\frac{\begin{array}{c}\emptyset \vdash k : B \qquad \emptyset \vdash t_2 : \mathsf{Bool} \qquad \vdash \{x{:}B \mid t_1\} : B \\ t_2 \longrightarrow_c^{*} \mathsf{true}\ \text{ implies }\ t_1\{x := k\} \longrightarrow_c^{*} \mathsf{true}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k\rangle^{l} : B}\ \text{T\_CHECKING}
$$

$$\boxed{\vdash c : T}$$

$$
\frac{x{:}B \vdash t : \mathsf{Bool}}{\vdash \{x{:}B \mid t\} : B}\ \text{T\_BASEC}
\qquad
\frac{\vdash c_1 : T_1 \qquad \vdash c_2 : T_2}{\vdash c_1 \mapsto c_2 : T_1 \to T_2}\ \text{T\_FUNC}
$$

Fig. 3. Syntax and semantics for λ_C

constants using two constructions: 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 $\mathscr{K}_B$); 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 $[\![k]\!](k_1) \in \mathscr{K}_{B_2}$ if $k_1 \in \mathscr{K}_{B_1}$.

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 3).

The evaluation contexts implement left-to-right evaluation for function application. If $\Uparrow l$ appears in the active position of an evaluation context, it is propagated to the top level, like an uncatchable exception. 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 when E_CONST reduces a well-typed application, the argument is not just a value, but a constant.

The rules E_CCHECK, E_OK, E_FAIL and E_CDECOMP, 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. We hold on to the original contract as a technical device for the translation ϕ from λ_C to λ_H, since λ_H needs to know the target type of an active check. 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. This is why function contracts applied to values are values. There is no substantial difference between this approach and expanding function contracts into new lambdas. 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 3) are mostly standard. 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 $\vdash c : T$, 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 considering only nondependent contracts for now. Contract application, like function application, is checked using T_APP.

The T_CHECKING rule only applies in the empty context (active checks are only created at the top level during evaluation). The rule ensures that the contract $\{x{:}B \mid t_1\}$ 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 implication: $t_2 \longrightarrow_c^* \mathsf{true}$ implies $t_1\{x := k\} \longrightarrow_c^* \mathsf{true}$ 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 Section 5). This rule obviously makes typechecking for the full "internal language" with checks undecidable, but excluding checks recovers decidability. We could give a more precise condition—for example, that $t_1\{x := k\} \longrightarrow_c^* t_2$—but there is no need.

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.

Syntax for λ_{H}

$$
\begin{array}{rcll}
S & ::= & \{x{:}B \mid s_1\} \mid S_1 \to S_2 & \text{types/contracts} \\[1ex]
s & ::= & x \mid k \mid \lambda x{:}S_1.\, s_2 \mid s_1\ s_2 \mid \Uparrow l \mid \langle S_1 \Rightarrow S_2 \rangle^l \mid \langle \{x{:}B \mid s_1\}, s_2, k\rangle^l & \text{terms} \\
w & ::= & k \mid \lambda x{:}S_1.\, s_2 \mid \langle S_1 \Rightarrow S_2 \rangle^l \mid \langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22} \rangle^l\ w & \text{values} \\
q & ::= & w \mid \Uparrow l & \text{results} \\
F & ::= & [\,]\, s \mid w\, [\,] \mid \langle \{x{:}B \mid s\}, [\,], k\rangle^l & \text{evaluation contexts}
\end{array}
$$

Operational semantics for λ_{H}

$$
\begin{array}{rcll}
(\lambda x{:}S_1.\, s_2)\ w_2 & \longrightarrow_h & s_2\{x := w_2\} & \text{F\_BETA} \\
k\ w & \longrightarrow_h & [\![k]\!](w) & \text{F\_CONST} \\
\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} \rangle^l\ k & \longrightarrow_h & \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l & \text{F\_CCHECK} \\
\langle \{x{:}B \mid s\}, \mathsf{true}, k\rangle^l & \longrightarrow_h & k & \text{F\_OK} \\
\langle \{x{:}B \mid s\}, \mathsf{false}, k\rangle^l & \longrightarrow_h & \Uparrow l & \text{F\_FAIL} \\
(\langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22} \rangle^l\ w)\ w' & \longrightarrow_h & \langle S_{12} \Rightarrow S_{22} \rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11} \rangle^l\ w')) & \text{F\_CDECOMP} \\
F[\Uparrow l] & \longrightarrow_h & \Uparrow l & \text{F\_BLAME} \\
F[s_1] & \longrightarrow_h & F[s_2] \quad \text{when } s_1 \longrightarrow_h s_2 & \text{F\_COMPAT}
\end{array}
$$

Typing rules for λ_{H}

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

$$\frac{x{:}S \in \Delta}{\Delta \vdash x : S}\ \text{S\_VAR} \qquad \frac{}{\Delta \vdash k : \mathsf{ty_h}(k)}\ \text{S\_CONST}$$

$$\frac{\vdash S_1 \qquad \Delta, x{:}S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x{:}S_1.\, s_2 : S_1 \to S_2}\ \text{S\_LAM} \qquad \frac{\Delta \vdash s_1 : S_1 \to S_2 \qquad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\ s_2 : S_2}\ \text{S\_APP}$$

$$\frac{\vdash S_1 \qquad \vdash S_2 \qquad \lfloor S_1 \rfloor = \lfloor S_2 \rfloor}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2}\ \text{S\_CAST} \qquad \frac{\vdash S}{\Delta \vdash \Uparrow l : S}\ \text{S\_BLAME}$$

$$\frac{\Delta \vdash s : S_1 \qquad \vdash S_2 \qquad \vdash S_1 <: S_2}{\Delta \vdash s : S_2}\ \text{S\_SUB}$$

$$\frac{\begin{array}{c}\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\} \qquad \emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \qquad \vdash \{x{:}B \mid s_1\} \\ s_2 \longrightarrow_h^* \mathsf{true} \ \text{ implies } \ s_1\{x := k\} \longrightarrow_h^* \mathsf{true}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid s_1\}, s_2, k\rangle^l : \{x{:}B \mid s_1\}}\ \text{S\_CHECKING}$$

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

$$\frac{\forall k \in \mathcal{K}_B.\ (s_1\{x := k\} \longrightarrow_h^* \mathsf{true} \ \text{ implies } \ s_2\{x := k\} \longrightarrow_h^* \mathsf{true})}{\vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}}\ \text{SSUB\_REFINE}$$

$$\frac{\vdash S_{21} <: S_{11} \qquad \vdash S_{12} <: S_{22}}{\vdash S_{11} \to S_{12} <: S_{21} \to S_{22}}\ \text{SSUB\_FUN}$$

$$\boxed{\vdash S}$$

$$\frac{}{\vdash \{x{:}B \mid \mathsf{true}\}}\ \text{SWF\_RAW} \qquad \underset{\text{SWF\_REFINE}}{\frac{x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\vdash \{x{:}B \mid s\}}} \qquad \frac{\vdash S_1 \qquad \vdash S_2}{\vdash S_1 \to S_2}\ \text{SWF\_FUN}$$

Fig. 4. Syntax and semantics for λ_{H}

The language λ_H

Our second core calculus, nondependent λ_{H}, extends the simply typed lambda-calculus with *refinement types* and *cast expressions*. The definitions appear in Figure 4. Unlike λ_{C}, which separates contracts from types, λ_{H} combines them into refined base types $\{x{:}B \mid s_1\}$ and function types $S_1 \to S_2$. As for λ_{C}, we do not allow refinement types over functions, nor do we allow refinements of refinements. (Belo *et al.* (2011) add these features to a dependent λ_{H}.) Unrefined base types B are *not* valid types; they must be wrapped in 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 term of type S_1 can be given type S_2. Like λ_{C}, active checks are used to give a small-step semantics to cast expressions.

The values of λ_{H} include constants, abstractions, casts, and function casts applied to values. Results are either values or blame. We give meaning to constants as we did in λ_{C}, reusing $[\![-]\!]$. Type assignment is via $\mathrm{ty_h}$, which we assume produces well-formed types (defined in Figure 4). To keep the languages in sync, we require that $\mathrm{ty_h}$ and $\mathrm{ty_c}$ agree on "type skeletons": if $\mathrm{ty_c}(k) = B_1 \to B_2$, then $\mathrm{ty_h}(k) = \{x{:}B_1 \mid s_1\} \to \{x{:}B_2 \mid s_2\}$.

The small-step, call-by-value semantics in Figure 4 comprises six basic rules and two rules involving evaluation contexts F. Each rule corresponds closely to its λ_{C} counterpart.

Notice how the decomposition rules compare. In λ_{C}, the term $(\langle c_1 \mapsto c_2\rangle^{l,l'}\ v)\ v'$ decomposes into two contract checks: 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 into two casts: a contravariant cast on the argument and a covariant cast on theresult. The contravariant cast $\langle S_{21} \Rightarrow S_{11}\rangle^l\ w'$ makes w' a suitable input for w, while $\langle S_{12} \Rightarrow S_{22}\rangle^l$ casts 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 carefully inspected to see exactly which checks will take place. On the other hand, which label will be blamed is clearer with casts—there's only one!

The typing rules for λ_{H} (Figure 4) 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 are well-formed and have the same simple-type skeleton, defined as $\lfloor - \rfloor : S \to T$ (pronounced "erase S"):

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

This prevents "stupid" casts, like $\langle \lceil \mathsf{Int} \rceil \Rightarrow \lceil \mathsf{Bool} \rceil\rangle^l$. We define a similar operator, $\lceil - \rceil : S \to S$ (pronounced "raw S"), which trivializes all refinements:

$$\begin{aligned}
\lceil \{x{:}B \mid s\} \rceil &= \{x{:}B \mid \mathsf{true}\} \\
\lceil S_1 \to S_2 \rceil &= \lceil S_1 \rceil \to \lceil S_2 \rceil
\end{aligned}$$

These operations apply to λ_C contracts and types in the natural way. Type well-formedness in λ_H is similar to contract well-formedness in λ_C, though the SWF_RAW case needs to be added to get things off the ground.

The active check rule S_CHECKING plays a role analogous to the T_CHECKING rule in λ_C, again using an implication to guarantee that we only have sensible terms in the predicate position. Note that we retain the target type in the active check, and that S_CHECKING gives active checks that type—technical moves necessary for preservation.

An important difference 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 contravariant subtyping rule. We do not consider source programs with subtyping, since subtyping makes type checking undecidable[2]; subtyping is just a technical device for ensuring type preservation. Consider the following reduction:

$$\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^l\ 1 \longrightarrow_h^* 1$$

The source term is well-typed at $\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$. Since it evaluates to 1, we would like to have $\Delta \vdash 1 : \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$. To have type preservation in general, though, $\mathrm{ty_h}(1)$ 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 way to satisfy this requirement is to set $\mathrm{ty_h}(k) = \{x{:}B \mid x = k\}$ for $k \in \mathscr{K}_B$; then if $s\{x := k\} \longrightarrow_h^* \mathsf{true}$, we have $\vdash \mathrm{ty_h}(k) <: \{x{:}B \mid s\}$. This approach is taken in Knowles & Flanagan (2010) and Ou *et al.* (2004).

Standard progress and preservation theorems hold for λ_H. We can also obtain a semantic type soundness theorem as a restriction of the one for dependent λ_H (Theorem 4.12).

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—positive and negative—where λ_H contracts have a single label. Finally, λ_H has a much richer type system than λ_C. 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 while carefully mapping "feature for feature".

The interesting parts of the translations deal with contracts and casts. Everything else is translated homomorphically, though the type annotation on lambdas must be chosen carefully. The full definitions of these translations are in Section 5; the nondependent definitions are a straightforward restriction.

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 translating the cast $\langle S_1 \Rightarrow S_2\rangle^l$ to a contract by translating the pair of types together, $\langle\psi(S_1,S_2)\rangle^{l,l}$. We define ψ as two mutually 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

[2] Flanagan (2006) and Knowles & Flanagan (2010) discuss trade-offs between static and dynamic checking that allow for decidable type systems and subtyping.

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, i.e.:

$$\psi(\langle S_1 \Rightarrow S_2\rangle^l) = \langle \psi(S_1, S_2)\rangle^{l,l}.$$

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,

$$\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \lceil \mathsf{Int} \rceil \Rightarrow \lceil \mathsf{Int} \rceil \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\ y\}\rangle^l$$

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

Translating from λ_C to λ_H, we are moving from a simple type system to a rich one. The translation ϕ (essentially the same as Gronski & Flanagan's) generates terms in λ_H with *raw* types—λ_H types with trivial refinements, corresponding to λ_C's simple types. 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 6.1).

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 is a kind of "bulletproofing." Bulletproofing maintains the raw-type invariant: the positive cast takes the argument out of $\lceil c \rceil$ and the negative cast returns it there. For example,

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

translates to the λ_H term

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

Unfolding the domain parts of the casts on f, the domain of the negative cast ensures 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 cast does nothing, since $\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \Rightarrow \lceil \mathsf{Int} \rceil\rangle^l$ has no effect. Similarly, the codomain of the negative cast does nothing while the codomain of the positive cast checks that the result is positive. Separating the checks allows λ_{H} to keep track of blame labels, mimicking λ_{C}. Put more generally, in the positive cast, the positive positions may fail because they are "down casts", whereas the negative positions are "up casts", so they cannot fail. The opposite is true of the negative cast. This embodies the idea of contracts as pairs of projections (Findler & Blume, 2006). Note that bulletproofing is "overkill" at base type: for example, $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}\rangle^{l,l'}$ translates to

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

Only the positive cast does anything—the negative cast into $\lceil \mathsf{Int} \rceil$ always succeeds. This asymmetry is consistent with λ_{C}, where base-type contracts also ignore the negative label. In Section 4 we extend the bulletproofing translation to dependent contracts—one of our main contributions.

Both ϕ and ψ 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. 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 SSUB_REFINE. To avoid a dangerous circularity, we define closing substitutions in terms of a separate type semantics. Additionally, the new F_CDECOMP rule has a slightly tricky (but necessary) asymmetry, explained below.

Dependent λ_C

Dependent λ_{C} has been studied since Findler & Felleisen (2002); it received a very thorough treatment (with an untyped host language) in Blume & McAllester (2006), was ported to Haskell by Hinze *et al.* (2006) and Chitil & Huch (2007), and was 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 & Felleisen (2002), with a few technical changes to make the proofs for ϕ easier.

We have marked the changed rules with a • next to their names. The new T_REFINEC, T_FUNC, and E_CDECOMP rules in Figure 5 suffice to add dependency to λ_{C}. To help us work with the translations, we also make some small changes to the bindings in contexts, adding a new binding form to track the labels on a contract check throughout the contract well-formedness judgment. Note that T_FUNC adds $x{:}{c_1}^{l',l}$ to the context when checking the codomain of a function contract, swapping blame labels. We add a new variable rule, T_VARC, that treats $x{:}c^{l,l'}$ as if it were its skeleton, $x{:}\lfloor c \rfloor$. While unnecessary for λ_{C}'s

Syntax for dependent λ_C

$$
\begin{array}{rcll}
T & ::= & B \mid T_1 \to T_2 & \text{types} \\
c & ::= & \{x{:}B \mid t\} \mid x{:}c_1 \mapsto c_2 & \text{contracts}\bullet \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T \mid \Gamma, x{:}c^{l,l'} & \text{typing contexts}\bullet \\
\\
t & ::= & x \mid k \mid \lambda x{:}T_1.\ t_2 \mid t_1\ t_2 \mid \Uparrow l \mid \langle c\rangle^{l,l'} \mid \langle \{x{:}B \mid t_1\}, t_2, k\rangle^{l} & \text{terms} \\
v & ::= & k \mid \lambda x{:}T_1.\ t_2 \mid \langle c\rangle^{l,l'} \mid \langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\ v & \text{values}\bullet \\
r & ::= & v \mid \Uparrow l & \text{results} \\
E & ::= & [\,]\ t \mid v\ [\,] \mid \langle \{x{:}B \mid t\}, [\,], k\rangle^{l} & \text{evaluation contexts}
\end{array}
$$

Operational semantics for λ_C

$$
\begin{array}{rcll}
(\lambda x{:}T_1.\ t_2)\ v & \longrightarrow_c & t_2\{x := v\} & \text{E\_BETA} \\
k\ v & \longrightarrow_c & [\![k]\!](v) & \text{E\_CONST} \\
\langle \{x{:}B \mid t\}\rangle^{l,l'}\ k & \longrightarrow_c & \langle \{x{:}B \mid t\}, t\{x := k\}, k\rangle^{l} & \text{E\_CCHECK} \\
(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\ v)\ v' & \longrightarrow_{lax} & \langle c_2\{x := v'\}\rangle^{l,l'}\ (v\ (\langle c_1\rangle^{l',l}\ v')) & \text{E\_CDECOMPLAX}\bullet \\
(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}\ v)\ v' & \longrightarrow_{picky} & \langle c_2\{x := \langle c_1\rangle^{l',l}\ v'\}\rangle^{l,l'}\ (v\ (\langle c_1\rangle^{l',l}\ v')) & \text{E\_CDECOMPPICKY}\bullet \\
\langle \{x{:}B \mid t\}, \mathsf{true}, k\rangle^{l} & \longrightarrow_c & k & \text{E\_OK} \\
\langle \{x{:}B \mid t\}, \mathsf{false}, k\rangle^{l} & \longrightarrow_c & \Uparrow l & \text{E\_FAIL} \\
E[\Uparrow l] & \longrightarrow_c & \Uparrow l & \text{E\_BLAME} \\
E[t_1] & \longrightarrow_c & E[t_2] \quad \text{when } t_1 \longrightarrow_c t_2 & \text{E\_COMPAT}
\end{array}
$$

Contract erasure

$$
\lfloor \{x{:}B \mid t\} \rfloor = B \qquad \lfloor x{:}c_1 \mapsto c_2 \rfloor = \lfloor c_1 \rfloor \to \lfloor c_2 \rfloor
$$

Typing rules for dependent λ_C

$$\boxed{\vdash \Gamma}$$

$$
\frac{}{\vdash \emptyset}\ \text{T\_EMPTY} \qquad
\frac{\vdash \Gamma}{\vdash \Gamma, x{:}T}\ \text{T\_EXTVART}\bullet \qquad
\frac{\vdash \Gamma}{\vdash \Gamma, x{:}T}\ \text{T\_EXTVART}\bullet
$$

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

$$
\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_VART} \qquad
\frac{x{:}c^{l,l'} \in \Gamma}{\Gamma \vdash x : \lfloor c \rfloor}\ \text{T\_VARC}\bullet \qquad
\frac{}{\Gamma \vdash k : \mathsf{ty_c}(k)}\ \text{T\_CONST}
$$

$$
\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2}\ \text{T\_LAM} \qquad
\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{T\_APP}
$$

$$
\frac{\Gamma \vdash^{l,l'} c : T}{\Gamma \vdash \langle c\rangle^{l,l'} : T \to T}\ \text{T\_CONTRACT}\bullet \qquad
\frac{}{\Gamma \vdash \Uparrow l : T}\ \text{T\_BLAME}
$$

$$
\frac{\begin{array}{c}\vdash \Gamma \qquad \emptyset \vdash k : B \qquad \emptyset \vdash t_2 : \mathsf{Bool} \qquad \emptyset \vdash^{l,l'} \{x{:}B \mid t_1\} : B \\ t_2 \longrightarrow_c^{*} \mathsf{true}\ \text{ implies }\ t_1\{x := k\} \longrightarrow_c^{*} \mathsf{true}\end{array}}{\Gamma \vdash \langle \{x{:}B \mid t_1\}, t_2, k\rangle^{l} : B}\ \text{T\_CHECKING}\bullet
$$

$$\boxed{\Gamma \vdash^{l,l'} c : T}$$

$$
\frac{\Gamma, x{:}B \vdash t : \mathsf{Bool}}{\Gamma \vdash^{l,l'} \{x{:}B \mid t\} : B}\ \text{T\_REFINEC}\bullet \qquad
\frac{\Gamma \vdash^{l',l} c_1 : T_1 \qquad \Gamma, x{:}c_1{}^{l',l} \vdash^{l,l'} c_2 : T_2}{\Gamma \vdash^{l,l'} x{:}c_1 \mapsto c_2 : T_1 \to T_2}\ \text{T\_FUNC}\bullet
$$

Fig. 5. Syntax and semantics for dependent λ_C

metatheory, this new binding form helps ϕ preserve types when translating from λ_{H} to picky λ_{C}; see Section 6.1.

Two different variants of the E_CDECOMP rule can be found in the literature: they are *lax* and *picky*. The original rule in Findler & Felleisen (2002) is lax (like most other contract calculi): it does not recheck c_1 when substituting v' into c_2. Blume & McAllester (2006) used a picky semantics without observing their departure from Findler and Felleisen; Hinze *et al.* (2006) choose to be picky as well, substituting $\langle c_1\rangle^{l',l}\ v'$ into c_2 because it makes their conjunction contract idempotent. We can show (straightforwardly) that both enjoy standard progress and preservation properties. Below, we consider translations to and from both dialects of λ_{C}: picky λ_{C} using only E_CDECOMPPICKY in Sections 5.1 and 6.2, and lax λ_{C} using only E_CDECOMPLAX in Sections 5.2 and 6.1. Accordingly, we give two sets of evaluation rules: $\longrightarrow_{lax}$ and $\longrightarrow_{picky}$. When we write $\longrightarrow_c$, the metavariable c ranges over *picky* and *lax*. We complete the type soundness proofs here generically, writing $\longrightarrow_c$ for the evaluation relation. For the translations in Section 4, we specify which evaluation relation we use.

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 typing derivation. □

For preservation, we prove confluence and substitution lemmas. Note that our substitution lemma must now also cover contracts, since they are no longer closed.

4.2 Lemma [Determinacy]: Let $\longrightarrow_c$ be either $\longrightarrow_{picky}$ or $\longrightarrow_{lax}$. If $t \longrightarrow_c t'$ and $t \longrightarrow_c t''$, then $t' = t'$.

4.3 Corollary [Coevaluation]: If Let $\longrightarrow_c$ be either $\longrightarrow_{picky}$ or $\longrightarrow_{lax}$. $t \longrightarrow_c^* r$ and $t \longrightarrow_c^* t'$, then $t' \longrightarrow_c^* r$.

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

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

Proof

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

We omit the proof for $x{:}c^{l,l'}$ bindings, which is similar.

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

Proof

By induction on the typing derivation. This proof is straightforward because typing and contracts hardly interact. □

Syntax for dependent λ_H

$$
\begin{array}{lcll}
S & ::= & \{x{:}B \mid s_1\} \mid x{:}S_1 \to S_2 & \text{types/contracts}\bullet \\
\Delta & ::= & \emptyset \mid \Delta, x{:}S & \text{typing contexts} \\
\\
s & ::= & x \mid k \mid \lambda x{:}S_1.\, s_2 \mid s_1\ s_2 \mid \ \mid \Uparrow l \mid \langle S_1 \Rightarrow S_2\rangle^l \mid \langle \{x{:}B \mid s_1\}, s_2, k\rangle^l & \text{terms} \\
w & ::= & k \mid \lambda x{:}S_1.\, s_2 \mid \langle S_1 \Rightarrow S_2\rangle^l \mid \langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\ w & \text{values}\bullet \\
q & ::= & w \mid \Uparrow l & \text{results} \\
F & ::= & [\,]\,s \mid w\,[\,] \mid \langle \{x{:}B \mid s\}, [\,], k\rangle^l & \text{evaluation contexts}
\end{array}
$$

Operational semantics for dependent λ_H

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

$$
\begin{array}{rcll}
(\lambda x{:}S_1.\, s_2)\ w_2 & \rightsquigarrow_h & s_2\{x := w_2\} & \text{F\_BETA} \\
k\ w & \rightsquigarrow_h & [\![k]\!](w) & \text{F\_CONST} \\
\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\ k & \rightsquigarrow_h & \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l & \text{F\_CCHECK} \\
(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\ w)\ w' & \rightsquigarrow_h & & \text{F\_CDECOMP}\bullet \\
\multicolumn{3}{r}{\langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\} \Rightarrow S_{22}\{x := w'\}\rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w'))} & \\
\langle \{x{:}B \mid s\}, \mathsf{true}, k\rangle^l & \rightsquigarrow_h & k & \text{F\_OK} \\
\langle \{x{:}B \mid s\}, \mathsf{false}, k\rangle^l & \rightsquigarrow_h & \Uparrow l & \text{F\_FAIL}
\end{array}
$$

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

$$
\frac{s_1 \rightsquigarrow_h s_2}{s_1 \longrightarrow_h s_2}\ \text{F\_REDUCE}
\qquad
\frac{s_1 \longrightarrow_h s_2}{F[s_1] \longrightarrow_h F[s_2]}\ \text{F\_COMPAT}
\qquad
\frac{}{F[\Uparrow l] \longrightarrow_h \Uparrow l}\ \text{F\_BLAME}
$$

Fig. 6. Syntax and operational semantics for dependent λ_H

Dependent λ_H

Now we come to the challenging part: dependent λ_H and its proof of type soundness. These results require the most complex metatheory in the paper because we need some strong properties about call-by-value evaluation.[3] The full definitions are in Figure 7. As before, we have marked the changed rules with a • next to their names.

We enrich the type system with dependent function types, $x{:}S_1 \to S_2$, where x may appear in S_2. The S_CAST rule and the proofs need a notion of type erasure, $\lfloor S \rfloor$; type height $|S|$ will also be used in the proofs.

$$
\begin{array}{rcl}
\lfloor - \rfloor & : & S \to T \\
\lfloor \{x{:}B \mid s\} \rfloor & = & B \\
\lfloor x{:}S_1 \to S_2 \rfloor & = & \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor
\end{array}
\qquad
\begin{array}{rcl}
| - | & : & S \to \mathbb{N} \\
|\{x{:}B \mid s\}| & = & 1 \\
|x{:}S_1 \to S_2| & = & 1 + |S_1| + |S_2|
\end{array}
$$

A new dependent application rule, S_APP, substitutes the argument into the result type of the application. We generalize SWF_REFINE to allow refinement-type predicates that use variables from the enclosing context. SWF_FUN adds the bound variable to the context when checking the codomain of function types. In SSUB_FUN, subtyping for dependent function types remains contravariant, but we also add the argument variable to the context

[3] The benefit of a CBV semantics is a better treatment of blame. By contrast, Knowles & Flanagan (2010) cannot treat failed casts as exceptions because that would destroy confluence. They treat them as stuck terms. Readers familiar with the soundness proof of Knowles & Flanagan (2010) will notice that our proof is significantly different from theirs. We discuss this in Section 7.

Typing rules

$\boxed{\vdash \Delta}$

$$\frac{}{\vdash \emptyset} \quad \text{S\_EMPTY} \qquad \frac{\vdash \Delta \quad \Delta \vdash S}{\vdash \Delta, x{:}S} \quad \text{S\_EXTVAR}$$

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

$$\frac{x{:}S \in \Delta}{\Delta \vdash x : S} \quad \text{S\_VAR} \qquad \frac{}{\Delta \vdash k : \mathsf{ty_h}(k)} \quad \text{S\_CONST}$$

$$\frac{\Delta \vdash S_1 \quad \Delta, x{:}S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x{:}S_1.\ s_2 : (x{:}S_1 \to S_2)} \quad \text{S\_LAM}\bullet \qquad \frac{\Delta \vdash s_1 : (x{:}S_1 \to S_2) \quad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\ s_2 : S_2\{x := s_2\}} \quad \text{S\_APP}\bullet$$

$$\frac{\Delta \vdash S_1 \quad \Delta \vdash S_2 \quad \lfloor S_1 \rfloor = \lfloor S_2 \rfloor}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2} \quad \text{S\_CAST} \qquad \frac{\Delta \vdash s : S_1 \quad \Delta \vdash S_2 \quad \Delta \vdash S_1 <: S_2}{\Delta \vdash s : S_2} \quad \text{S\_SUB}\bullet$$

$$\frac{\vdash \Delta \quad \emptyset \vdash k : \{x{:}B \mid \mathsf{true}\} \quad \emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \quad \emptyset \vdash \{x{:}B \mid s_1\} \quad \emptyset \vdash s_2 \supset s_1\{x := k\}}{\Delta \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}} \quad \text{S\_CHECKING}\bullet$$

$\boxed{\Delta \vdash S}$

$$\frac{}{\Delta \vdash \{x{:}B \mid \mathsf{true}\}} \ \text{SWF\_RAW} \qquad \frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\Delta \vdash \{x{:}B \mid s\}} \ \text{SWF\_REFINE}\bullet \qquad \frac{\Delta \vdash S_1 \quad \Delta, x{:}S_1 \vdash S_2}{\Delta \vdash x{:}S_1 \to S_2} \ \text{SWF\_FUN}\bullet$$

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

$$\frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 \supset s_2}{\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}} \ \text{SSUB\_REFINE}\bullet \qquad \frac{\Delta \vdash S_{21} <: S_{11} \quad \Delta, x{:}S_{21} \vdash S_{12} <: S_{22}}{\Delta \vdash x{:}S_{11} \to S_{12} <: x{:}S_{21} \to S_{22}} \ \text{SSUB\_FUN}\bullet$$

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

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

$$\Delta \models \sigma \iff \forall x \in \mathrm{dom}(\Delta).\ \sigma(x) \in [\![\sigma(\Delta(x))]\!]$$

Fig. 7. Typing rules for dependent λ_{H}

with the smaller type. This is similar to the function subtyping rule of $F_{<:}$ (Cardelli *et al.*, 1994).

We need to be careful when implementing higher-order dependent casts in the rule F_CDECOMP. As the cast decomposes, the variables in the codomain types of such a cast must be replaced. However, this substitution is asymmetric; on one side, we cast the argument and on the other we do not. This behavior is required for type soundness. 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 $\Delta \vdash (\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} \rangle^l\ w)\ w'$:

$S_{22}\{x := w'\}$. 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 $\Delta \vdash w\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w') : S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\}$; in order to apply the codomain cast to this term, we must substitute $\langle S_{21} \Rightarrow S_{11}\rangle^l\ w'$ into S_{12}. This calculation determines the form of F_CDECOMP.

While the operational semantics changes only in F_CDECOMP, we have 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).

The final change generalizes SSUB_REFINE to open terms. We must close these terms before we can compare their behavior, using closing substitutions σ and reading $\Delta \models \sigma$ as "σ satisfies Δ".

Care is needed here to prevent the typing rules from becoming circular: the typing rule S_SUB references the subtyping judgment, the subtyping rule SSUB_REFINE references the implication judgment, and the single implication rule S_IMP has $\Delta \models \sigma$ in a negative position. This circularity would cause the typing rules to be non-monotonic, and so the existence of a least or greatest fixed-point would not be immediately obvious—our type system would not be well defined! To avoid this circularity, $\Delta \models \sigma$ must not refer back to the other judgments. (The reader may wonder why this was not a problem in λ_{C}, but notice that in λ_{C}, implication is only used in T_CHECKING—which has no (real) context. If we only needed implication in the S_CHECKING rule, we would not need contexts here, either—we can ensure that active checks only occur at the top-level, with an empty context. But the SSUB_REFINE subtyping rule refers to S_IMP, and subtyping may be used in arbitrary contexts.)

We can avoid the circularity and ensure that the type system is well defined by building the syntactic rules on top of a denotational semantics for λ_{H}'s types.[4] The idea is that the semantics of a type is a set of closed terms 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 strictly 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 8. It is defined by induction on 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 yield well-typed results. By construction, these sets include only terminating terms that do not get stuck. In order to show that casts inhabit the denotations of their types, we must also

[4] Knowles & Flanagan (2010) also introduce a type semantics, but theirs 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. The problem is not insurmountable: either step indexing their semantics or a proof of unwinding as in Pitts (2005) would resolve the issue.

Denotations of types

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

Denotations of kinds

$$
\begin{array}{rcl}
\{x{:}B \mid s\} \in [\![\star]\!] & \iff & \forall k \in \mathscr{K}_B.\ s\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!] \\
x{:}S_1 \to S_2 \in [\![\star]\!] & \iff & S_1 \in [\![\star]\!] \wedge \forall q \in [\![S_1]\!].\ S_2\{x := q\} \in [\![\star]\!]
\end{array}
$$

Semantic judgments

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

Fig. 8. Type and kind semantics for dependent λ_{H}

define a denotation of kinds. Since the only kind is $*$, its denotation $[\![\star]\!]$ directly defines semantic well-formedness in terms of the denotations of types.

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 require that constants are semantically well typed: $k \in [\![\mathrm{ty_h}(k)]\!]$; this requirement is true by our "most-specific type" assumption at base types, but must be assumed at (first-order) functions types. Note that this rules out including fix as a constant, since our type semantics is inhabited only by strongly normalizing terms. We conjecture that expanding the denotation of refinement types to allow for divergence or a step-indexed logical relation (Ahmed, 2006) would allow us to consider nonterminating terms.

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

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

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

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

Proof
By induction on $|S|$. $\square$

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

Proof
By induction on $|S|$. $\square$

4.10 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 $\Uparrow l$.

4.11 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 $|S|$.

$\underline{S = \{x{:}B \mid s_0\}}$: Suppose $s \in [\![\{x{:}B \mid s_0\}]\!]$. By definition, either $s \longrightarrow_h^* w$ or $s \longrightarrow_h^* \Uparrow l$, so s normalizes.

$\underline{S = x{:}S_1 \to S_2}$: 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 4.10), 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. If the application reduces to a value (i.e., $s\ q \longrightarrow_h^* w$), then first $s\ q \longrightarrow_h^* w'\ q$, and so $s \longrightarrow_h^* w'$. Alternatively, the application could reduce to blame (i.e., $s\ q \longrightarrow_h^* \Uparrow l$). 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 both cases, s normalizes. $\square$

Unlike the rest of the paper, we take a top-down approach to the rest of type soundness, to help motivate the steps. We are interested in relating our syntactic type system and the type semantics by *semantic type soundness*: if $\emptyset \vdash s : S$, then $s \in [\![S]\!]$. However, to prove this result, we must generalize it. In the bottom of Figure 8, 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 terms belong to the type semantics.) We then show that the typing judgments imply their semantic counterparts.

4.12 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. $\square$

The first part follows by induction on the subtyping judgment.

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

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>SSub_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:

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

We must show $\Delta \models \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}$, i.e., that $\forall \sigma.\ (\Delta \models \sigma \ \text{ implies } \ [\![\{x{:}B \mid s_1\}]\!] \subseteq [\![\{x{:}B \mid s_2\}]\!])$. 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 $k \in \mathscr{K}_B$ 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 $k \in \mathscr{K}_B$, 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 4.13, $k \in [\![\{x{:}B \mid \mathsf{true}\}]\!]$.

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>SSUB_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 $s \in [\![\sigma(x{:}S_{21} \to S_{22})]\!]$. $\square$

The proof semantic subtype soundess goes through easily, the first of the three parts of semantic soundness (Theorem 4.12). We run into some complications with semantic type and kind soundness, 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 8, 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 is not necessarily a result! We do know, however, that $s_2 \in [\![S_1]\!]$, so $s_2 \longrightarrow_h^* q_2$ by strong normalization (Lemma 4.11). 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 9 that allows redexes 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, though we need to be careful to make it respect our call-by-value reduction order: the β-redex $(\lambda x{:}S_1.\, s_1)\ s_2$ should not be contracted unless s_2 is a value, since doing so can change the order of effects. (Other redices within s_1 and s_2 can safely reduce.)[5] The proof requires a longish sequence of technical lemmas that essentially show that $\Rrightarrow$ commutes with $\longrightarrow_h^*$. Since the proofs require fussy symbol manipulation, we have done these proofs in Coq. Our development 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 $s_2 \longrightarrow_h^* \Uparrow l$.

[5] We conjecture that the reflexive transitive closure of 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.

$$\boxed{s_1 \Rrightarrow s_2}$$

$$\frac{}{s \Rrightarrow s}\ \text{FP\_REFL} \qquad \frac{w \Rrightarrow w'}{k\ w \Rrightarrow [\![k]\!](w')}\ \text{FP\_RCONST} \qquad \frac{s_{12} \Rrightarrow s'_{12} \quad w_2 \Rrightarrow w'_2}{(\lambda x{:}S.\ s_{12})\ w_2 \Rrightarrow s'_{12}\{x := w'_2\}}\ \text{FP\_RBETA}$$

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

$$\frac{}{\langle \{x{:}B \mid s\}, \mathsf{true}, k\rangle^l \Rrightarrow k}\ \text{FP\_ROK} \qquad \frac{}{\langle \{x{:}B \mid s\}, \mathsf{false}, k\rangle^l \Rrightarrow \Uparrow l}\ \text{FP\_RFAIL}$$

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

$$\frac{S_1 \Rrightarrow S'_1 \quad s_{12} \Rrightarrow s'_{12}}{\lambda x{:}S_1.\ s_{12} \Rrightarrow \lambda x{:}S'_1.\ s'_{12}}\ \text{FP\_LAM} \qquad \frac{s_1 \Rrightarrow s'_1 \quad s_2 \Rrightarrow s'_2}{s_1\ s_2 \Rrightarrow s'_1\ s'_2}\ \text{FP\_APP} \qquad \frac{S_1 \Rrightarrow S'_1 \quad S_2 \Rrightarrow S'_2}{\langle S_1 \Rightarrow S_2\rangle^l \Rrightarrow \langle S'_1 \Rightarrow S'_2\rangle^l}\ \text{FP\_CAST}$$

$$\frac{S \Rrightarrow S' \quad s \Rrightarrow s'}{\langle S, s, k\rangle^l \Rrightarrow \langle S', s', k\rangle^l}\ \text{FP\_CHECK} \qquad \frac{}{F[\Uparrow l] \Rrightarrow \Uparrow l}\ \text{FP\_BLAME}$$

$$\boxed{S_1 \Rrightarrow S_2}$$

$$\frac{}{S \Rrightarrow S}\ \text{FP\_SREFL} \qquad \frac{s \Rrightarrow s'}{\{x{:}B \mid s\} \Rrightarrow \{x{:}B \mid s'\}}\ \text{FP\_SREFINE} \qquad \frac{S_1 \Rrightarrow S'_1 \quad S_2 \Rrightarrow S'_2}{x{:}S_1 \to S_2 \Rrightarrow x{:}S'_1 \to S'_2}\ \text{FP\_SFUN}$$

Fig. 9. Parallel reduction for dependent λ_H

An alternative strategy would be to use $\Rrightarrow$ in the typing rules and $\longrightarrow_h$ in the operational semantics. This would 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 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 $S_1 \Rrightarrow S_2$. $\square$

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 $\Delta_1, x{:}S', \Delta_2 \models s : S$, and $\Delta_1, x{:}S', \Delta_2 \models S$, 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 the definition of $\Delta \models \sigma$. $\square$

The semantic typing case for casts requires a separate induction.

4.20 Lemma [Semantic typing for casts]: If $\Delta \models S_1$ and $\Delta \models S_2$ 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 $\sigma(\langle S_1 \Rightarrow S_2 \rangle^l) \in [\![\sigma(S_1 \to S_2)]\!]$.
$\underline{S_2 = \{x{:}B \mid \sigma(s_2)\}}$: Let $q \in [\![\sigma(S_1)]\!]$. If $q = \Uparrow l'$, then the applied cast goes to $\Uparrow l'$, and we are done by Lemma 4.9. 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 4.11), 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 4.8) completes the proof.
$\underline{S_2 = x{:}S_{21} \to S_{22}}$: We must have $S_1 = x{:}S_{11} \to S_{12}$. Let $q \in [\![\sigma(S_1)]\!]$; if it is blame we are done by Lemma 4.9, so let it be a value w. Let $q' \in [\![\sigma(S_{21})]\!]$; if it is blame we are 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 4.11), 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. Using parallel reduction (Corollary 4.18), we have $[\![\sigma(S_{12})\{x := q''\}]\!] = [\![\sigma(S_{12})\{x := \langle \sigma(S_{21}) \Rightarrow \sigma(S_{11}) \rangle^l\ w'\}]\!]$.

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

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

4.21 Lemma [Semantic type soundness]:

1. If $\Delta \vdash s : S$ then $\Delta \models s : S$.
2. If $\Delta \vdash S$ then $\Delta \models S$.

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. $\square$

Theorem 4.12 gives us type soundness, and it combines with Lemma 4.11 for an even stronger result: well-typed programs always evaluate to values of appropriate (semantic) type.

While one can prove progress and preservation theorems, we omit them: we already have type soundness. Our later proofs will require standard weakening and substitution lemmas, though, so we prove them now.

4.22 Lemma [Weakening]: If $\Delta \vdash s : S$ and $\Delta \vdash S$, and $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset$ with $\vdash \Delta, \Delta'$, then $\Delta, \Delta' \vdash s : S$ and $\Delta, \Delta' \vdash S$.

Proof
By straightforward induction on s and $|S|$; we reuse the (critical) context well-formedness derivation in the S_CHECKING case. □

The substitution lemma 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.

4.23 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, unfolding the closing substitutions. □

4.24 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. □

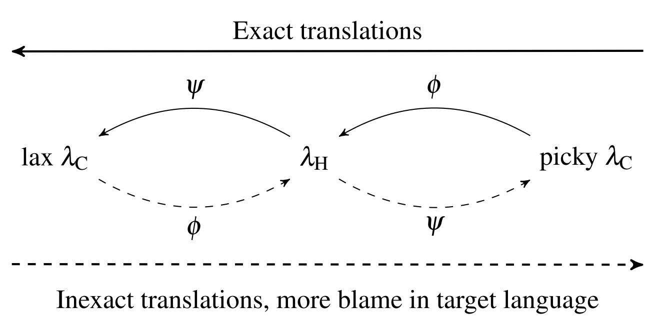
4.25 Lemma [Substitution (typing and well-formedness)]: If $\Delta_1 \vdash s : S$ then

1. if $\Delta_1, x{:}S, \Delta_2 \vdash s_1 : S_1$ then $\Delta_1, \Delta_2\{x := s\} \vdash s_1\{x := s\} : S_1\{x := s\}$,
2. if $\Delta_1, x{:}S, \Delta_2 \vdash S_1$ then $\Delta_1, \Delta_2\{x := s\} \vdash S_1\{x := s\}$, and
3. if $\vdash \Delta_1, x{:}S, \Delta_2$ then $\vdash \Delta_1, \Delta_2\{x := s\}$.

Proof
By mutual induction on the typing derivations. □

The translations

We divide our treatment of the translations between lax λ_C, λ_H, and picky λ_C into two sections: one for exact translations, moving right on the axis of blame, and one for inexact translations, moving left.



Section 5 covers the exact translations, moving left on the axis of blame from picky λ_C to λ_H, and from λ_H to lax λ_C. Section 6 covers the inexact translations, moving right on the axis of blame from lax λ_C to λ_H, and from λ_H to picky λ_C.

Each translation proof follows the same basic schema. First, we define a logical relation between the two languages. Then we use the logical relation to prove a lemma relating the

Result correspondence

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

$$k \approx k : B \iff k \in \mathscr{K}_B$$

$$v \approx w : T_1 \to T_2 \iff \forall t \sim s : T_1.\, v\, t \sim w\, s : T_2$$

$$\Uparrow l \approx \Uparrow l : T$$

Term correspondence

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

$$t \sim s : T \iff t \longrightarrow_c^* r \wedge s \longrightarrow_h^* q \wedge r \approx q : T$$

Fig. 10. A blame-exact result/term correspondence

translation, contracts, and casts. Finally, we prove that the translation preserves evaluation behavior—that is, terms are logically related to their translations—and typing. All of the proofs make extensive use of expansion and contraction of evaluation and "cotermination" arguments. Every proof uses its own contract/cast logical relation. The proofs for the inexact translations of Section 6 demand custom term logical relations, too. We've used σ to range over closing substitutions in λ_{H}; we will use δ to range over dual closing substitutions in the logical relations.

5 Exact translations

Translations moving left on the axis of blame—from picky λ_{C} to λ_{H}, and from λ_{H} to lax λ_{C}—are exact. That is, we can show a tight behavioral correspondence between terms and their translations (see Figure 10). We read $t \sim s : T$ as "t corresponds with s at type T".

Our correspondence is a standard logical relation, defined in two intertwined parts: a relation on results, $r \approx q : T$ and its closure with respect to evaluation, $t \sim s : T$. The term correspondence is defined directly: terms correspond when they reduce to corresponding results. We write $\longrightarrow_c$ in this single definition: in Section 5.1 we use this definition taking $\longrightarrow_c$ to be $\longrightarrow_{picky}$; in Section 5.2 we use this definition taking $\longrightarrow_c$ to be $\longrightarrow_{lax}$. The result correspondence is defined inductively over λ_{C}'s simple types. Blame corresponds to itself at any type. At B, constants in $\mathscr{K}_B$ correspond to themselves; results at $T_1 \to T_2$ correspond when they applying them to corresponding *terms* yields corresponding *terms*. Stratifying the definition this way simplifies some of our proofs later. We call this correspondence *exact* because terms corresponding at base type yield identical results.

Note that we define the correspondence here on closed (or harmlessly open) terms. In the following two sections, we'll define translation specific extensions of the correspondence to open terms and contracts.

5.1 Translating picky λ_C to λ_H: dependent ϕ

We define the full ϕ for the dependent calculi in Figure 11. 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 translate derivations to ensure type preservation, translating T_VART and T_VARC derivations differently: we leave variables of simple type alone, but we cast variables bound to contracts.

To see why we need 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

$$
\begin{array}{lrcl}
\textbf{Contexts} & \phi : (\vdash \Gamma) \to \Delta & & \\
& \phi(\vdash \emptyset) & = & \emptyset \\
& \phi(\vdash \Gamma, x{:}T) & = & \phi(\vdash \Gamma), x{:}\lceil T \rceil \\
& \phi(\vdash \Gamma, x{:}c^{l,l'}) & = & \phi(\vdash \Gamma), x{:}\phi(\Gamma \vdash^{l,l'} c : \lfloor c \rfloor)
\end{array}
$$

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

$$
\begin{array}{lrcl}
\textbf{Types} & \phi : (\Gamma \vdash^{l,l'} c : T) \to S & & \\
& \phi(\Gamma \vdash^{l,l'} \{x{:}B \mid t\} : B) & = & \{x{:}B \mid \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})\} \\
& \phi(\Gamma \vdash^{l,l'} x{:}c_1 \mapsto c_2 : T_1 \to T_2) & = & x{:}\phi(\Gamma \vdash^{l',l} c_1 : T_1) \to \phi(\Gamma, x{:}{c_1}^{l',l} \vdash^{l,l'} c_2 : T_2)
\end{array}
$$

Fig. 11. The translation ϕ from dependent λ_C to dependent λ_H

codomain "abuses" the bound variable. A naive translation will *not* be well-typed in λ_H. The term $f\ 0$ will not be typeable when f has type $x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \to \lceil \mathsf{Int} \rceil$, since f only accepts positive arguments. The problem is that SWF_FUN can add a (possibly refined) type to the context when checking the codomain, so we need to restore the "variables have raw types" invariant—something we can't always rely on subtyping to do, since types are not in general subtypes of their raw type. 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. We therefore translate the contract above to $f{:}S \to \{z{:}\mathsf{Int} \mid (\langle S \Rightarrow \lceil \mathsf{Int} \to \mathsf{Int} \rceil \rangle^{l'}\ f)\ 0 = 0\}$, where $S = x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \to \lceil \mathsf{Int} \rceil$. This (partially) motivates the $x{:}c^{l,l'}$ binding form in dependent λ_C.

Bulletproofing uses raw types, defined here for the dependent system.

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

Note that dependency is eliminated.

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: 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 first-order constant functions. This slightly restricts the types we might assign to our constants, e.g., we cannot say $\mathrm{ty_h}(\mathsf{sqrt}) = x{:}\{x{:}\mathsf{Float} \mid x >= 0\} \to \{y{:}\mathsf{Float} \mid$

Contract/type correspondence

$$\boxed{c \sim^{l,l'} S : T}$$

$$\{x{:}B \mid t\} \sim^{l,l'} \{x{:}B \mid s\} : B \iff \forall k \in \mathcal{K}_B.\ t\{x := k\} \sim s\{x := k\} : \mathsf{Bool}$$

$$\begin{array}{rcl} x{:}c_1 \mapsto c_2 \sim^{l,l'} x{:}S_1 \to S_2 : T_1 \to T_2 & \iff & c_1 \sim^{l',l} S_1 : T_1 \wedge \\ & & \forall t \sim s : T_1 . c_2\{x := \langle c_1 \rangle^{l',l}\ t\} \sim^{l,l'} S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ s\} : T_2 \end{array}$$

Dual closing substitutions

$$\Gamma \models \delta \iff \left\{ \begin{array}{c} \forall x{:}T \in \Gamma.\ \delta_1(x) \sim \delta_2(x) : T \\ \forall x{:}c^{l,l'} \in \Gamma.\ \delta_1(x) = \langle \delta_1(c) \rangle^{l,l'}\ t \wedge \delta_2(x) = \langle \lceil c \rceil \Rightarrow \delta_2(S) \rangle^{l}\ s \\ \text{where } S = \phi(\Gamma \vdash^{l,l'} c : \lfloor c \rfloor) \wedge t \sim s : \lfloor c \rfloor \end{array} \right.$$

Lifted to open terms

$$\begin{array}{rcl} \Gamma \vdash t \sim s : T & \iff & \forall \delta.\ (\Gamma \models \delta \text{ implies } \delta_1(t) \sim \delta_2(s) : T) \\ \Gamma \vdash c \sim^{l,l'} S : T & \iff & \forall \delta.\ (\Gamma \models \delta \text{ implies } \delta_1(c) \sim^{l,l'} \delta_2(S) : T) \end{array}$$

Fig. 12. Blame-exact correspondence for ϕ from picky λ_C

$(y * y) = x\}$, since it is not the case that $\Delta \vdash \mathrm{ty_h}(\mathsf{sqrt}) <: \lceil \mathsf{Float} \to \mathsf{Float} \rceil$. 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 have already required that denotations be total over their simple types in λ_C, and λ_H uses the same denotation function $[\![-]\!]$, so this requirement does not seem too severe. In any case, we can define it to be equal to $\Uparrow l_0$, for some 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 extend the term correspondence of Figure 10 to contracts and types, lifting the correspondences to open terms using dual closing substitutions. Recall that we interpret the term correspondence as using $\longrightarrow_{picky}$. For a binding $x{:}c^{l,l'} \in \Gamma$, we use ϕ to insert the negative cast (labelled with l') and closing substitutions (in Figure 12) to insert the positive cast (labelled with l). Do not be confused by the label used for function contract correspondence—this definition does, in fact, match up with closing substitutions. A binding $x{:}c^{l,l'} \in \Gamma$ must have come from the domain of an application of T_FUNC, so the labels on the binding are *already* swapped when ϕ or $\Gamma \models \delta$ sees them. In the definition of function contract correspondence, we swap manually—whence the l' on the inserted cast. It helps to think of polarity in terms of position rather than the presence or absence of a prime.

5.1 Lemma [Expansion and contraction]: If $t \longrightarrow_{picky}{}^{*} t'$, and $s \longrightarrow_h^* s'$ then $t \sim s : T$ iff $t' \sim s' : T$.

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

5.3 Lemma [Equivalence is closed under parallel reduction]: If $s \Rrightarrow s'$ then $t \sim s : T$ iff $t \sim s' : T$. Similarly, if $S \Rrightarrow S'$ then $c \sim^{l,l'} S : T$ iff $c \sim^{l,l'} S' : T$.

Proof

In both cases, by induction on T, using the first to prove the second. $\square$

5.4 Lemma [Trivial casts]: If $t \sim s : B$ and $\lfloor S \rfloor = B$, then $t \sim \langle S \Rightarrow \lceil B \rceil \rangle^{l}\ s : B$.

5.5 Lemma [Related base casts]: If $\{x{:}B \mid t\} \sim^{l_0,l_1} \{x{:}B \mid s\} : B$ and $t' \sim s' : B$ and $\lfloor S \rfloor = B$, then $\langle\{x{:}B \mid t\}\rangle^{l,l'}\ t' \sim \langle S \Rightarrow \{x{:}B \mid s\}\rangle^{l}\ s' : B$.

Proof

Direct. Note that l_0 and l_1 are entirely irrelevant. □

5.6 Lemma [Bulletproofing]: If $t \sim s : T$ and $c \sim^{l,l'} S : T$ 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. First, observe that either both t and s go to $\Uparrow l''$ or both t and s go to values related at T. In the former case, the outer terms also go to blame. So we only consider the case where $t \longrightarrow_{picky}{}^{*}\ v$, $s \longrightarrow_h^{*} w$, and $v \approx w : T$.

$\underline{T = B}$: So $c = \{x{:}B \mid t_1\}$ and $S = \{x{:}B \mid s_1\}$ and $S' = \{x{:}B \mid s_2\}$. By Lemma 5.5 we have $\langle c\rangle^{l,l'}\ t \sim \langle \lceil S\rceil \Rightarrow S\rangle^{l}\ s : B$. By Lemma 5.4 we can add the extra, trival cast $\langle S \Rightarrow \lceil S\rceil\rangle^{l'}$.

$\underline{T = T_1 \to T_2}$: We know that $c = x{:}c_1 \mapsto c_2$ and $S = x{:}S_1 \to S_2$. Let $t' \sim s' : T_1$. We only need to consider the case where $t' \longrightarrow_{picky}{}^{*}\ v'$ and $s' \longrightarrow_h^{*} w'$—if $t' \longrightarrow_{picky}{}^{*} \Uparrow l''$ and $s' \longrightarrow_h^{*} \Uparrow l''$ the outer terms correspond because both blame l''.

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

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

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?

— $T_1 = B$. Since $v' \approx w' : B$, we must have $v' = w' = k \in \mathcal{K}_B$. By Lemma 5.5 and $c_1 \sim^{l',l} S_1 : B$, we know that $\langle c_1\rangle^{l',l}\ v' \sim \langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w' : B$. Both terms goes to blame or to the same value—which must be k, from inspection of the contract and cast evaluation rules.. The former case is immediate, since the outer terms then go to blame. So suppose $\langle c_1\rangle^{l',l}\ k \longrightarrow_{picky}{}^{*}\ k$ and $\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k \longrightarrow_h^{*} k$. Now the terms evaluate like so:

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

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

By Lemma 5.4, $k \sim \langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ k : B$, so $v\ k \sim w\ (\langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ k) : T_2$.
We have by definition (and $k \sim k : B$) that $c_2\{x := \langle c_1\rangle^{l',l}\ k\} \sim^{l,l'} S_2\{x := \langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k\} : T_2$. Recall that $\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k \longrightarrow_h^{*} k$. This implies $\langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k \Rrightarrow^{*} k$ (Lemma `A4` in the Coq). We can then see that $S_2\{x := \langle \lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k\} \Rrightarrow^{*}$

$S_2\{x := k\}$ by Lemma `A1` in the Coq. By extension with the congruence rules:

$$\begin{array}{l}\langle S_2\{x := \langle\lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k\} \Rightarrow \lceil S_2\rceil\rangle^{l'} \\ \qquad \langle\lceil S_2\rceil \Rightarrow S_2\{x := \langle\lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k\}\rangle^{l}\ (w\ (\langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ k)) \Rrightarrow \\ \langle S_2\{x := \langle\lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ k\} \Rightarrow \lceil S_2\rceil\rangle^{l'} \\ \qquad \langle\lceil S_2\rceil \Rightarrow S_2\{x := k\}\rangle^{l}\ (w\ (\langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ k))\end{array}$$

By the IH $\langle c_2\{x := \langle c_1\rangle^{l',l}\ k\}\rangle^{l,l'}\ (v\ k)$ corresponds to the former, which means it is related to the latter by Lemma 5.3. We conclude the case with expansion (Lemma 5.1).

— $T_1 = T_{11} \to T_{12}$. We continue with an application of F_CDECOMP in λ_{H}:

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

By the IH on $c_1 \sim^{l',l} S_1 : T_1$ and $v' \sim w' : T_1$, we can find what we need for the domain: $\langle c_1\rangle^{l',l}\ v' \sim \langle S_1 \Rightarrow \lceil S_1\rceil\rangle^{l}\ (\langle\lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w') : T_1$. 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 have $c_2\{x := \langle c_1\rangle^{l',l}\ v'\} \sim^{l,l'} S_2\{x := \langle\lceil S_1\rceil \Rightarrow S_1\rangle^{l'}\ w'\} : T_2$ by assumption, so the IH tells us that the codomain contract and bulletproofing correspond. We conclude by expansion (Lemma 5.1). □

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

5.7 Theorem [Behavioral correspondence]: If $\vdash \Gamma$, then:

1. If $\phi(\Gamma \vdash t : T) = s$ then $\Gamma \vdash t \sim s : T$.
2. If $\phi(\Gamma \vdash^{l,l'} c : T) = S$ then $\Gamma \vdash c \sim^{l,l'} S : T$.

Proof
We simultaneously show both properties by induction on the depth of ϕ's recursion. □

We can now prove that ϕ preserves types, using Theorem 5.7 to show that ϕ preserves the implication judgment. As a preliminary, we use the behavioral correspondence to show that ϕ preserves the implication judgment.

5.8 Lemma: If $t_1 \longrightarrow_{picky}{}^{*}$ true implies $t_2 \longrightarrow_{picky}{}^{*}$ true then $\emptyset \vdash \phi(\emptyset \vdash t_1 : \mathsf{Bool}) \supset \phi(\emptyset \vdash t_1 : \mathsf{Bool})$.

Proof
By the logical relation. □

The type preservation proof is very similar to the correspondence proof of Theorem 5.7.

5.9 Theorem [Type preservation]: If $\phi(\vdash \Gamma) = \Delta$, then:

1. $\vdash \Delta$.

Term translation

$$\boxed{\psi : s \to t}$$

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

Cast translation

$$\boxed{\psi : S \times S \times l \to T}$$

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

Fig. 13. ψ mapping dependent λ_{H} to dependent λ_{C}

2. If $\phi(\Gamma \vdash t : T) = s$ then $\Delta \vdash s : \lceil T \rceil$.
3. If $\phi(\Gamma \vdash^{l,l'} c : T) = S$ 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. □

5.2 Translating λ_H to lax λ_C: dependent ψ

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

The full definition of ψ is in Figure 13. Most terms are translated homomorphically. In abstractions, the annotation is translated by erasing the refined λ_{H} type to its skeleton. As we mentioned in Section 3, the trickiest part is the translation of casts between function types: when generating the codomain contract from a cast between two function types, we perform the same asymmetric substitution as F_CDECOMP. Since ψ inserts new casts, we need to pick a blame label: $\psi(\langle S_1 \Rightarrow S_2 \rangle^l)$ passes l as an index to $\psi^l(S_1, S_2)$.

We reuse the term correspondence $t \sim s : T$ (Figure 10), interpreting it as using $\longrightarrow_{lax}$. and define a new contract/cast correspondence $c \sim S_1 \Rightarrow^l S_2 : T$ (Figure 14), relating contracts and pairs of λ_{H} types—effectively, casts. This correspondence uses the term correspondence in the base type case and follows the pattern of F_CDECOMP in the function case. Since it inserts a cast in the function case, we index the relation 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; we lift the relation to open terms in the standard way.

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

5.10 Lemma [Expansion and contraction]: If $t \longrightarrow_{lax}{}^{*} t'$, and $s \longrightarrow_h^{*} s'$ then $t \sim s : T$ iff $t' \sim s' : T$.

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

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

Contract/type correspondence

$$\boxed{c \sim S_1 \Rightarrow^l S_2 : T}$$

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

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

Dual closing substitutions

$$\Gamma \models \delta \iff \begin{cases} \forall x{:}T \in \Gamma, & \delta_1(x) \sim \delta_2(x) : T \\ \forall x{:}c^{l,l'} \in \Gamma, & \delta_1(x) \sim \delta_2(x) : \lfloor c \rfloor \end{cases}$$

Lifted to open terms

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

$$\Gamma \vdash c \sim S_1 \Rightarrow^l S_2 : T \iff \forall \delta.\ (\Gamma \models \delta \text{ implies } \delta_1(c) \sim \delta_2(S_1) \Rightarrow^l \delta_2(S_2) : T)$$

Fig. 14. Blame-exact correspondence for ψ into lax λ_C

As a corollary of Lemma 5.11 and Lemma 5.10, 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.

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

5.14 Lemma [Contract/cast correspondence]: If $c \sim S_1 \Rightarrow^l S_2 : T$ 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 5.10), showing that the initial terms reduce to corresponding terms.

$\underline{T = B}$: 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 $t \longrightarrow_{lax}{}^* k$ along with $s \longrightarrow_h^* k$.
We can step our terms into active checks as follows, then:

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

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.
$\underline{T = T_1 \to T_2}$: $c = x{:}c_1 \mapsto c_2$, $S_1 = x{:}S_{11} \to S_{12}$, and $S_2 = x{:}S_{21} \to S_{22}$. We know by inversion of the contract/cast relation that $c_1 \sim S_{21} \Rightarrow^l S_{11} : T_1$ and that for all $t \sim s : T_1$, $c_2\{x := t\} \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ s\} \Rightarrow^l S_{22}\{x := s\} : T_2$. 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 $t \longrightarrow_{lax}{}^* v$ 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

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

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 $\langle c_2\{x := v'\}\rangle^{l,l}\ (v\ (\langle c_1 \rangle^{l,l}\ v')) \sim \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w'\} \Rightarrow S_{22}\{x := w'\}\rangle^{l}\ w\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w') : T_2$. $\square$

We prove three more technical lemmas necessary for the behavioral and type correspondence.

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

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

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

We use the correspondence relations to show that s and its translation $\psi(s)$ correspond—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 "extra" substitution that ψ does. Instead, we do induction on the depth of ψ's recursion, (and also derivation height, for the S_SUB case).

5.18 Theorem [Behavioral correspondence]:

1. If $\Delta \vdash s : S$ then $\lfloor \Delta \rfloor \vdash \psi(s) \sim s : \lfloor S \rfloor$.
2. If $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = \lfloor S \rfloor$, then $\lfloor \Delta \rfloor \vdash \psi^{l}(S_1, S_2) \sim S_1 \Rightarrow^{l} S_2 : \lfloor S \rfloor$ (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.

We give only the most interesting cases for the first part: S_CAST, S_CHECKING, and S_SUB.

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), $\lfloor \Delta \rfloor \vdash \psi^l(S_1, S_2) \sim S_1 \Rightarrow^l S_2 : \lfloor S_2 \rfloor$.

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 5.14.

S_CHECKING: We have $\Delta \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}$; translating yields $\psi(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) = \langle \{x{:}B \mid \psi(s_1)\}, \psi(s_2), k \rangle^l$. Recall that the terms of the active check are closed. 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 5.15, $\lfloor S' \rfloor = \lfloor S \rfloor$.

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 are done.

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

$\psi^l(S_1, \{x{:}B \mid s_2\}) = \{x{:}B \mid \psi(s_2)\}$: Note that $S_2 = \{x{:}B \mid s_2\}$. 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 $\lfloor \Delta \rfloor \vdash \{x{:}B \mid \psi(s_2)\} \sim S_1 \Rightarrow^l \{x{:}B \mid s_2\} : B$. Let $\lfloor \Delta \rfloor \models \delta$; we prove that $\delta_1(\{x{:}B \mid \psi(s_2)\}) \sim \delta_2(S_1) \Rightarrow^l \delta_2(\{x{:}B \mid s_2\}) : B$, 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:

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

We can apply the IH (contravariantly) to see

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

By weakening (Lemma 4.22), 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 $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash \psi^l(S_{21}, S_{11}) \sim S_{21} \Rightarrow^l S_{11} : \lfloor S_{11} \rfloor$. 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 5.17, 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

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

We now combine (*) and (**) to show $\lfloor \Delta \rfloor \vdash \psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) \sim x{:}S_{11} \to S_{12} \Rightarrow^l x{:}S_{21} \to S_{22} : \lfloor S_2 \rfloor$. Let $\lfloor \Delta \rfloor \models \delta$. We can apply (*) to see $\delta_1(\psi^l(S_{21}, S_{11})) \sim$

$\delta_2(S_{21}) \Rightarrow^l \delta_2(S_{11}) : \lfloor S_{11} \rfloor$. For the codomain we must show, for all $t \sim s : \lfloor S_{11} \rfloor$, that

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

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

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

which we can rewrite to

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

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.

5.19 Lemma: If $\emptyset \vdash s_1 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash s_1 \supset s_2$, then $\psi(s_1) \longrightarrow_{lax}{}^* \mathsf{true}$ implies $\psi(s_2) \longrightarrow_{lax}{}^* \mathsf{true}$.

Proof
By the logical relation. □

The type preservation proof is very similar to the correspondence proof of Theorem 5.18, though the function case of the type/contract correspondence is intricate.

5.20 Theorem [Type preservation for ψ]:

1. If $\Delta \vdash s : S$ then $\lfloor \Delta \rfloor \vdash \psi(s) : \lfloor S \rfloor$.
2. If $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = T$, then $\lfloor \Delta \rfloor \vdash^{l,l'} \psi^l(S_1, S_2) : T$.

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

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

We must have $S_2 = x{:}S_{21} \to S_{22}$ 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 $\lfloor \Delta \rfloor \vdash^{l',l} \psi^l(S_{21}, S_{11}) : \lfloor S_{11} \rfloor$. 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 $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash^{l',l} \psi^l(S_{21}, S_{11}) : \lfloor S_{11} \rfloor$. 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 5.17, 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 $\Delta, x{:}S_{21} \vdash S_{22}$,

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

By Lemma 5.16, $\lfloor \psi^l(S_{21}, S_{11}) \rfloor = \lfloor S_{21} \rfloor$, so we can rewrite the above derivation to

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

Now by T_FUNC

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

6 Inexact translations

The same translations ϕ and ψ can be used to move right on the axis of blame (Figure 1). However, in this direction the images of these translations blame strictly more than their pre-images. We were able to use the same correspondence for both exact proofs in Section 5, but the following two proofs use custom correspondences: one where lax λ_C terms correspond to λ_H terms (with possibly more blame), and one where λ_H terms correspond to picky λ_C terms (with possibly more blame). In both cases, the λ_C terms will be on the left and the λ_H terms on the right.

6.1 Translating lax λ_C to λ_H

Translating with ϕ from terms in picky λ_C to exactly corresponding terms in λ_H was a relatively straightforward generalization of the nondependent case; things get more interesting when we consider the translation ϕ from lax λ_C to dependent λ_H. We can prove that it preserves types (for terms without active checks), but we can only show a weaker behavioral correspondence: sometimes lax λ_C terms terminate with values when their ϕ-images go to blame. This weaker property is a consequence of bulletproofing, the asymmetrically substituting F_CDECOMP rule, and the extra casts inserted for type preservation (i.e., for T_VARC derivations). This is not a weakness of our proof technique—we give a counterexample, a lax λ_C term $\emptyset \vdash t : T$ such that $t \longrightarrow_{lax}{}^* v$ and $\phi(\emptyset \vdash t : T) \longrightarrow_h^* \Uparrow l$.

We can show the behavioral correspondence using a blame-inexact logical relation, defined in Figure 15. The behavioral correspondence here, though weaker than before, is still pretty strong: if $t \sim_{\succ} s : B$ (read "t blames no more than s at type B"), then either $s \longrightarrow_h^* \Uparrow l$ or t and s both go to $k \in \mathcal{K}_B$. This correspondence differs slightly in construction from the earlier exact one—we define $\approx_{\succ}$ as a relation on *values*, while $\approx$ is a relation on *results*. Doing so simplifies our inexact treatment of blame—in particular, Lemma 6.2. We again use the term correspondence to relate contracts and λ_H types. We then lift the correspondences to open terms (Figure 15). Closing substitutions map variables to corresponding terms of appropriate type. Note that closing substitutions ignore the contract part of $x{:}c^{l,l'}$ bindings, treating them as if they were $x{:}\lfloor c \rfloor$.

6.1 Lemma [Expansion and contraction]: If $t \longrightarrow_{lax}{}^* t'$, and $s \longrightarrow_h^* s'$ then $t \sim_{\succ} s : T$ iff $t' \sim_{\succ} s' : T$.

Note that there are corresponding terms at every type. We can prove a much stronger lemma than we did for $\sim$ in Lemma 5.11, since the correspondence here is much weaker.

Value correspondence

$$\boxed{v \approx_{\succ} w : T}$$

$$k \approx_{\succ} k : B \iff k \in \mathscr{K}_B$$

$$v \approx_{\succ} w : T_1 \to T_2 \iff \forall t \sim_{\succ} s : T_1.\ v\ t \sim_{\succ} w\ s : T_2$$

Term correspondence

$$\boxed{t \sim_{\succ} s : T}$$

$$t \sim_{\succ} s : T \iff s \longrightarrow_h^* \Uparrow l \vee (t \longrightarrow_{lax}{}^* v \wedge s \longrightarrow_h^* w \wedge v \approx_{\succ} w : T)$$

Contract/type correspondence

$$\boxed{c \sim_{\succ} S : T}$$

$$\{x{:}B \mid t\} \sim_{\succ} \{x{:}B \mid s\} : B \iff \forall k \in \mathscr{K}_B.\ t\{x := k\} \sim_{\succ} s\{x := k\} : \mathsf{Bool}$$

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

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

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

Fig. 15. Blame-inexact correspondence for ϕ from lax λ_{C}

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

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

Proof
By induction on $\mathrm{ty_c}(k)$, recalling that constants are first order. □

As a corollary of Lemma 6.2 and Lemma 6.1, 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.

6.4 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.

6.5 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.

The third lemma shows that correspondence is closed under adding extra casts to the λ_{H} term, due to the inexactness of our behavioral correspondence. Since λ_{H} terms can go to blame more often than corresponding lax λ_{C} terms, we can add "extra" casts to λ_{H} terms. We formalize this in the following lemma, which captures the asymmetric treatment of blame by the $\sim_{\succ}$ relation. We use it to show that the cast substituted in the codomain by F_CDECOMP 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 witness the well-formedness of the λ_H types.

6.6 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.

$\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = T$. Either $s \longrightarrow_h^* \Uparrow l'$ or t and s both go to corresponding values 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 6.2).

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

$\underline{T = B}$: So $S_2 = \{x{:}B \mid s_2\}$, and $c_2 = \{x{:}B \mid t_2\}$.
So $t \longrightarrow_{lax}{}^* k$ 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 $k' \in \mathcal{K}_{\mathsf{Bool}}$. 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 (by Lemma 6.2). In the latter case, the λ_H term either goes to $\Uparrow l$ (and everything is related to blame) or goes to k—but so does t, and $k \approx_\succ k : B$.
$\underline{T = T_1 \to T_2}$: We have:

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

We have $t \longrightarrow_{lax}{}^* v$ 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 $v\ t' \sim_\succ (\langle S_1 \Rightarrow S_2 \rangle^l\ w)\ s' : T_2$. 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 6.2, so let $t' \longrightarrow_{lax}{}^* v'$ and $s' \longrightarrow_h^* w'$.
Decomposing the cast in λ_H,

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

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 6.1). $\square$

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.

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

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

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 when $T = B \to T_2$.

6.8 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 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 6.2). So $t \longrightarrow_{lax}{}^* v$, $s \longrightarrow_h^* w$, and $v \approx_{\succ} w : T$.

<u>$T = B$</u>: So $c = \{x{:}B \mid t_1\}$ and $S = \{x{:}B \mid s_1\}$ and $S' = \{x{:}B \mid s_2\}$. By Lemma 6.5 we have $\langle c \rangle^{l,l'}\ t \sim_{\succ} \langle \lceil S \rceil \Rightarrow S \rangle^{l}\ s : B$. By Lemma 6.4 we can add the extra, trivial cast.
<u>$T = T_1 \to T_2$</u>: 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 6.2, we only need to consider the case where $t' \longrightarrow_{lax}{}^* v'$ and $s' \longrightarrow_h^* w'$—if $s' \longrightarrow_h^* {\Uparrow} l''$ we are done.
On the λ_C side, $(\langle c \rangle^{l,l'}\ t)\ t' \longrightarrow_{lax}{}^* \langle c_2\{x := v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v'))$. 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 6.5 and $c_1 \sim_{\succ} S_1' : B$, we know that $\langle c_1 \rangle^{l',l}\ v' \sim_{\succ} \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w' : B$. The λ_H term goes to blame or both terms go to the same value, $v' = w' = k \in \mathscr{K}_B$. In the former case, the entire λ_H term goes to blame and we are done by Lemma 6.2. So suppose $\langle c_1 \rangle^{l',l}\ k \longrightarrow_{lax}{}^* k$ and $\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w' \longrightarrow_h^* k$. Now the terms evaluate like so:

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

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

By Lemma 6.4, $k \sim_{\succ} \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k : B$, so $v\ k \sim_{\succ} w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k) : T_2$.

Noting that $k \rightsquigarrow k : B$ and $k \rightsquigarrow \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^l\, k : B$, we can see that $c_2\{x := k\} \rightsquigarrow S_2\{x := k\} : T_2$ and $c_2\{x := k\} \rightsquigarrow S_2'\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^l\, k\} : T_2$. Now the IH shows that $\langle c_2\{x := k\}\rangle^{l,l'}\,(v\,k) \rightsquigarrow \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\, w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'}\, \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\}\rangle^l\,(w\,(\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^l\, k)) : T_2$, and we conclude the case with expansion (Lemma 6.1).

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

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

By the IH, $\langle c_1 \rangle^{l',l}\, v' \rightsquigarrow \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^l\, \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\, w' : T_1$. 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 $c_1 \rightsquigarrow S_1' : T_1$, and by Lemma 6.7 $T_1 {\uparrow} \rightsquigarrow \lceil S_1' \rceil : T_1$. Since $v' \rightsquigarrow w' : T_1$, Lemma 6.6 shows $v' \rightsquigarrow \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\, w' : T_1$. This lets us see that $c_2\{x := v'\} \rightsquigarrow S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\, w'\} : T_2$ and $c_2\{x := v'\} \rightsquigarrow S_2\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\, w'\} : T_2$. Now the IH and expansion (Lemma 6.1) complete the proof. □

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

6.9 Theorem [Behavioral correspondence]: If $\vdash \Gamma$, then:

1. If $\phi(\Gamma \vdash t : T) = s$ then $\Gamma \vdash t \rightsquigarrow s : T$.
2. If $\phi(\Gamma \vdash^{l,l'} c : T) = S$ then $\Gamma \vdash c \rightsquigarrow S : T$.

Proof

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

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

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

We can also show type preservation for terms not containing active checks. (We do not know that translated active checks are well typed, because Theorem 6.9 is not strong enough to preserve the implication judgment. We only expect these checks to occur at runtime, so this is good enough: ϕ preserves the types of source programs.)

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

1. $\vdash \Delta$.
2. $\Delta \vdash \phi(\Gamma \vdash t : T) : \lceil T \rceil$.
3. $\Delta \vdash \phi(\Gamma \vdash^{l,l'} c : T)$.

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. $\square$

To see that the ϕ in Figure 11 does not give us exact blame, let us look at two counterexamples; in both cases, a lax λ_{C} term goes to a value while its translation goes to blame. In the first example, blame is raised in λ_{H} due to bulletproofing. In the second, blame is raised due to the extra cast from the translation of T_VARC. In both examples, the contracts are *abusive*: higher-order contracts where the codomain places a contradictory requirement on the domain. For the first counterexample, let

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

Here, the contradiction comes when the codomain requires that $f\ 0$ yield 0, but f's contract says it will return a non-zero value. We find $\langle c\rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_{lax}{}^{*} 0$ but

$$(\lambda x{:}\lceil c\rceil.\ \langle S \Rightarrow \lceil S\rceil\rangle^l\ (\langle \lceil S\rceil \Rightarrow S\rangle^l\ x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l.$$

For the second counterexample, let

$$\begin{array}{rcl}
c' & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\} \\
S'_1 & = & x{:}\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\} \\
S' & = & \phi(\emptyset \vdash^{l,l} c' : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \\
 & = & f{:}S'_1 \to \{z{:}\mathsf{Int} \mid (\langle S'_1 \Rightarrow \lceil S'_1 \rceil\rangle^l\ f)\ 0 = 0\}.
\end{array}$$

This time, the contradiction comes from the codomain applying f to 0, while the domain contract requires that f's input be non-zero. We find $\langle c'\rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_{lax}{}^{*} 0$ but

$$(\lambda x{:}\lceil c'\rceil.\ \langle S' \Rightarrow \lceil c'\rceil\rangle^l\ (\langle \lceil S\rceil \Rightarrow \lceil c'\rceil\rangle^l\ x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l.$$

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*.

6.2 Translating λ_H to picky λ_C

Terms in λ_{H} and their ψ-images in lax λ_{C} correspond exactly, as shown Section 5.2. When we change the operational semantics of λ_{C} to be *picky*, however, $\psi(s)$ blames (strictly) more often than s. Nevertheless, we can show an inexact correspondence, as we did for ϕ and lax λ_{C} in Section 6.1. We use a logical relation $\sim_{\prec}$ (similar to our earlier inexact relation) for ϕ into lax λ_{C} (Figure 15). Here we have reversed the asymmetry: picky λ_{C} may blame more than λ_{H}. The proof follows the same general pattern: we first show that it is safe to add extra contract checks, then we show that contracts and casts correspond (inexactly), then the correspondence for well-typed terms. We can also show type preservation for source programs (excluding active checks).

Value correspondence

$$\boxed{v \approx_{\prec} w : T}$$

$$k \approx_{\prec} k : B \iff k \in \mathscr{K}_B$$

$$v \approx_{\prec} w : T_1 \to T_2 \iff \forall t \sim_{\prec} s : T_1.\ v\, t \sim_{\prec} w\, s : T_2$$

Term correspondence

$$\boxed{t \sim_{\prec} s : T}$$

$$t \sim_{\prec} s : T \iff t \longrightarrow_{picky}{}^{*} \Uparrow l \lor t \longrightarrow_{picky}{}^{*} v \land s \longrightarrow_h^{*} w \land v \approx_{\prec} w : T$$

Contract/type correspondence

$$\boxed{c \sim_{\prec} S_1 \Rightarrow S_2 : T}$$

$$\{x{:}B \mid t\} \sim_{\prec} \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} : B \iff \forall k \in \mathscr{K}_B.\ t\{x := k\} \sim_{\prec} s_2\{x := k\} : \mathsf{Bool}$$

$$x{:}c_1 \mapsto c_2 \sim_{\prec} x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} : T_1 \to T_2 \iff c_1 \sim_{\prec} S_{21} \Rightarrow S_{11} : T_1 \land \forall l. \forall t \sim_{\prec} s : T_1.\ c_2\{x := t\} \sim_{\prec} S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^{l}\, s\} \Rightarrow S_{22}\{x := s\} : T_2$$

Dual closing substitutions

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

Lifted to open terms

$$\Gamma \vdash t \sim_{\prec} s : T \iff \forall \delta.\ (\Gamma \models \delta \text{ implies } \delta_1(t) \sim_{\prec} \delta_2(s) : T)$$

$$\Gamma \vdash c \sim_{\prec} S_1 \Rightarrow S_2 : T \iff \forall \delta.\ (\Gamma \models \delta \text{ implies } \delta_1(c) \sim_{\prec} \delta_2(S_1) \Rightarrow \delta_2(S) : T)$$

Fig. 16. Blame-inexact correspondence for ψ into picky λ_{C}

6.11 Lemma [Expansion and contraction]: If $t \longrightarrow_{picky}{}^{*} t'$ and $s \longrightarrow_h^{*} s'$ then $t \sim_{\prec} s : T$ iff $t' \sim_{\prec} s' : T$.

6.12 Lemma [Blame corresponds to everything]: For all T, $\Uparrow l \sim_{\prec} s : T$.

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

Proof

By induction on $\mathrm{ty_c}(k)$, recalling that constants are first order. □

As a corollary of Lemma 6.12 and Lemma 6.11, if a picky λ_{C} term evaluates to blame, then it corresponds to any λ_{H} term. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

6.14 Lemma [Extra contracts]: If $t \sim_{\prec} s : T$ and $c \sim_{\prec} S_1 \Rightarrow S_2 : T$ then $\langle c\rangle^{l,l'}\, t \sim_{\prec} s : T$.

Proof

By induction on T. If $t \longrightarrow_{picky}{}^{*} \Uparrow l''$ we are done, so let $t \longrightarrow_{picky}{}^{*} v$ and $s \longrightarrow_h^{*} w$ such that $v \approx_{\prec} w : T$.

<u>$T = B$</u>: So $c = \{x{:}B \mid t_2\}$ and $S_2 = \{x{:}B \mid s_2\}$. Moreover, $v = w = k \in \mathscr{K}_B$, since those are the only corresponding values at B.
We can step and see $\langle c\rangle^{l,l'}\, t \longrightarrow_{picky}{}^{*} \langle c, t_2\{x := k\}, k\rangle^{l}$. We know that $t_2\{x := k\} \sim_{\prec} s_2\{x := k\} : \mathsf{Bool}$. There are two possibilities: either $t_2\{x := k\} \longrightarrow_{picky}{}^{*} \Uparrow l''$ or both

terms go to corresponding Bools. In the former case, the whole λ_C term goes to blame and we are done by Lemma 6.12. If both go to false, then the outer λ_C term evaluates to $\Uparrow l$ and we are done by Lemma 6.12 again. If both go to true, then both outer terms go to k, and $k \approx_{\prec} k : B$.

$\underline{T = T_1 \to T_2}$: So $c = x{:}c_1 \mapsto c_2$ and $S_1 = x{:}S_{11} \to S_{12}$ and $S_2 = x{:}S_{21} \to S_{22}$. Let $t' \sim_{\prec} s' : T_1$. If $t' \longrightarrow_{picky}{}^* \Uparrow l''$ we are done by Lemm 6.12, so let $t' \longrightarrow_{picky}{}^* v'$ and $s' \longrightarrow_h^* w'$, where $v' \approx_{\prec} w' : T_1$. We want to prove $(\langle c \rangle^{l,l'}\ t)\ t' \sim_{\prec} s\ s' : T_2$, which is true iff:

$$\langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\}\rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \sim_{\prec} w\ w' : T_2$$

By the IH on $v \sim_{\prec} w' : T_1$ and $c_1 \sim_{\prec} S_{21} \Rightarrow S_{11} : T_1$, we have $\langle c_1 \rangle^{l',l}\ v' \sim_{\prec} w' : T_1$. By definition, applying v and w yields related terms at T_2. Since $\langle c_1 \rangle^{l',l}\ v' \sim_{\prec} w' : T_1$, we have $c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \sim_{\prec} S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w'\} \Rightarrow S_{22}\{x := w'\} : T_2$. We can now apply the IH and see:

$$\langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\}\rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \sim_{\prec} w\ w' : T_2 \quad \square$$

6.15 Lemma [Contract/cast correspondence]: If $c \sim_{\prec} S_1 \Rightarrow S_2 : T$ and $t \sim_{\prec} s : T$ then $\langle c \rangle^{l,l'}\ t \sim_{\prec} \langle S_1 \Rightarrow S_2 \rangle^{l''}\ s : T$.

Proof

By induction on T. We reason via expansion (Lemma 6.11), showing that the initial terms reduce to corresponding terms.

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

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

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

By the contract/cast correspondence, we know that $t_1\{x := k\} \sim_{\prec} s_2\{x := k\} : \mathsf{Bool}$, so either $t_1\{x := k\}$ goes to blame or both terms go to a Bool together. In the former case, the outer λ_C term goes to blame and we are done by Lemma 6.12.. If they go to false, then both the active check goes to $\Uparrow l$ and we are done, again by Lemma 6.12. Finally, if they both go to true, then both terms will evaluate to $k \in \mathcal{K}_B$, and $k \approx_{\prec} k : B$.

$\underline{T = T_1 \to T_2}$: $c = x{:}c_1 \mapsto c_2$, $S_1 = x{:}S_{11} \to S_{12}$, and $S_2 = x{:}S_{21} \to S_{22}$. We know by inversion of the contract/cast relation that $c_1 \sim_{\prec} S_{21} \Rightarrow S_{11} : T_1$ and that for all l'' and $t \sim_{\prec} s : T_1$, $c_2\{x := t\} \sim_{\prec} S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ s\} \Rightarrow S_{22}\{x := s\} : T_2$. We want to prove that $\langle c \rangle^{l,l} \sim_{\prec} \langle S_1 \Rightarrow S_2 \rangle^{l}\ s : T_1 \to T_2$. First, we can assume $t \longrightarrow_{picky}{}^* v$ and $s \longrightarrow_h^* w$ where $v \sim_{\prec} w : T_1 \to T_2$—if not, both the contracted term goes to blame and we are done by Lemma 6.12.

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

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

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 $\langle c_1\rangle^{l',l}\ v' \sim_{\prec} \langle S_{21} \Rightarrow S_{11}\rangle^{l''}\ w' : T_1$. Since $v \sim_{\prec} w : T_1 \to T_2$, we have $v\ (\langle c_1\rangle^{l',l}\ v') \sim_{\prec} w\ (\langle S_{21} \Rightarrow S_{11}\rangle^{l''}\ w') : T_2$. By Lemma 6.14, $\langle c_1\rangle^{l',l}\ v' \sim_{\prec} w' : T_1$. We can then see that $c_2\{x := \langle c_1\rangle^{l',l}\ v'\} \sim_{\prec} S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^{l''}\ w'\} \Rightarrow S_{22}\{x := w'\} : T_2$. By the IH, we therefore have

$$\langle c_2\{x := \langle c_1\rangle^{l',l}\ v'\}\rangle^{l,l'}\ (v\ (\langle c_1\rangle^{l',l}\ v')) \sim_{\prec} \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^{l''}\ w'\} \Rightarrow S_{22}\{x := w'\}\rangle^{l''}\ w\ (\langle S_{21} \Rightarrow S_{11}\rangle^{l''}\ w') : T_2 \quad \square$$

6.16 Theorem [Behavioral correspondence]:

1. If $\Delta \vdash s : S$ then $\lfloor\Delta\rfloor \vdash \psi(s) \sim_{\prec} s : \lfloor S\rfloor$.
2. If $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor = \lfloor S\rfloor$, then $\lfloor\Delta\rfloor \vdash \psi^l(S_1,S_2) \sim_{\prec} S_1 \Rightarrow S_2 : \lfloor S\rfloor$.

Proof
By an induction similar to the proof Theorem 5.18. $\square$

6.17 Theorem [Type preservation for ψ]: For programs without active checks, if $\vdash \Delta$, then:

1. If $\Delta \vdash s : S$ then $\lfloor\Delta\rfloor \vdash \psi(s) : \lfloor S\rfloor$.
2. If $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor = T$, then $\lfloor\Delta\rfloor \vdash^{l,l'} \psi^l(S_1,S_2) : T$.

Proof
By an induction similar to the proof Theorem 5.20. $\square$

Here is an example where a λ_{H} term reduces to a value while its ψ-image in picky λ_{C} term reduces to blame. As before, this counterexample uses an *abusive* contract: a higher-order contract where the codomain puts a contradictory requirement on the domain. Here, the contradiction is that f claims to return a non-zero value, but the codomain requires that it return 0.

$$\begin{array}{rcl} S_1 & = & f{:}S_{11} \to S_{12} \\ & = & f{:}(x{:}\lceil\mathsf{Int}\rceil \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}) \to \lceil\mathsf{Int}\rceil \\ S_2 & = & f{:}S_{21} \to S_{22} \\ & = & f{:}(x{:}\lceil\mathsf{Int}\rceil \to \lceil\mathsf{Int}\rceil) \to \{z{:}\mathsf{Int} \mid f\ 0 = 0\} \\ c & = & \psi^l(S_1,S_2) \\ & = & f{:}\psi^l(S_{21},S_{11}) \mapsto \psi^l(S_{12}\{f := \langle S_{21} \Rightarrow S_{11}\rangle^l f\}, S_{22}) \\ & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\} \end{array}$$

Let $w = (\lambda f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}).\ 0)$ and $w' = (\lambda x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\}.\ 0)$. The term is well typed: we can show $\emptyset \vdash w : S_1$ and $\emptyset \vdash w' : S_{21}$. Therefore $\emptyset \vdash (\langle S_1 \Rightarrow S_2\rangle^l\ w)\ w' : S_{22}\{f := w'\}$. Translating, we find

$$\psi((\langle S_1 \Rightarrow S_2\rangle^l\ w)\ w') = (\langle \psi^l(S_1,S_2)\rangle^{l,l}\ \psi(w))\ \psi(w') = (\langle c\rangle^{l,l}\ \lambda f{:}\mathsf{Int}.\ 0)\ \lambda x{:}\mathsf{Int}.\ 0.$$

On the one hand $(\langle S_1 \Rightarrow S_2\rangle^l\ w)\ w' \longrightarrow_h^* 0$, while $(\langle c\rangle^{l,l}\ \lambda f{:}\mathsf{Int}.\ 0)\ \lambda x{:}\mathsf{Int}.\ 0 \longrightarrow_{picky}{}^* \Uparrow l$. This means we cannot hope to use ψ as an exact correspondence between λ_{H} and picky λ_{C}.

(Removing the extra cast ψ inserts into S_{12} does not affect our example, since ψ ignores S_{12} here.) For example,

$$\psi^l(\{z{:}\mathsf{Int} \mid \mathsf{true}\}\{f := \langle S_{21} \Rightarrow S_{11}\rangle^l f\}, \{z{:}\mathsf{Int} \mid f\ 0 = 0\}) = \{x{:}B \mid \psi(f\ 0 = 0)\}.$$

6.3 Alternative calculi

There are three alternative calculi we have not considered here: indy λ_C (Dimoulas *et al.*, 2011), superpicky λ_H, and nonterminating calculi. We describe them in detail below, but we leave them as future work.

Dimoulas *et al.* (2011) add a third blame label to λ_C, representing the contract itself; we write it here as a subscript. They accordingly change the picky E_CDECOMP rule:

$$(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}_{l''}\ v_1)\ v_2 \longrightarrow_{indy} \langle c_2\{x := \langle c_1\rangle^{l'',l}_{l''}\ v_2\}\rangle^{l,l'}_{l''}\ (v_1\ (\langle c_1\rangle^{l',l}_{l''}\ v_2))$$

In the substitution in the codomain, note that the blame label on the domain contract uses the contract's blame label l''. The intuition here is that any problem arising in c_2 is in the contract's context (label l''), not the original negative context (label l'). We conjecture (but have not proven) that indy λ_C is in the same position on the axis of blame as picky λ_C. We should only need to change the labels on the contracts ϕ inserts to have an exact correspondence; however, ψ will remain inexact.

Superpicky λ_H reworks the F_CDECOMP rule in an attempt to harmonize λ_H and picky λ_C semantics:[6]

$$\begin{array}{l}(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\ w_1)\ w_2 \rightsquigarrow_h \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w_2\} \Rightarrow S_{22}\{x := \langle S_{11} \Rightarrow S_{21}\rangle^l\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w_2)\}\rangle^l \\ \qquad (w_1\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w_2))\end{array}$$

This seems to resolve the problem with ψ into picky λ_C, but it poses problems in the proof of semantic type soundness for λ_H: how do $S_{22}\{x := w_2\}$ and $S_{22}\{x := \langle S_{11} \Rightarrow S_{21}\rangle^l\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w_2)\}$ relate?

Finally, we have been careful to ensure that all of our calculi are strongly normalizing. We do not believe this to be essential, though we would have to change our logical relations—λ_H's type semantics and the correspondences—to account for nontermination. We conjecture that step-indexing (Ahmed, 2006) will suffice.

7 Related work

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 Table 1 we summarize the important points of comparison between a number of systems that are closely related to ours. This table is an updated version compared to that in Greenberg *et al.* (2010).

[6] This idea is due to Jeremy Siek (personal communication, January 2010).

Table 1. *Comparison between contract systems*

Latent systems

	FF02 (1)	HJL06 (2)	GF07 λ_C (3)	BM06 (4)	DFFF11 (5)	our λ_C
dep (6)	✓ lax	✓ picky	×	(7)	✓ indy	✓ either
eval order	CBV	lazy	CBV	CBV	CBV	CBV
blame (8)	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$ or $\perp$	$\Uparrow l$	$\Uparrow l$
checking (9)	if	if	$\bigcirc$	active	active	active
typing (10)	✓	✓	✓	n/a	✓	✓
any con (11)	✓	✓	✓	✓	✓	✓

Manifest systems

	GF07 λ_H (3)	F06 (12)	KF10 (13)	WF09 (14)	OTMW04 (15)	BGIP11 (16)	our λ_H
dep (6)	×	✓	✓	×	✓	✓	✓
eval order	CBV	CBN(17)	full β	CBV	CBV	CBV	CBV
blame (8)	$\Uparrow l$	stuck	stuck	$\Uparrow l$	$\Uparrow$	$\Uparrow l$	$\Uparrow l$
checking (9)	$\bigcirc$	$\bigcirc$	active	active	if	active	active
typing (10)	×	×	✓	✓	✓	✓	✓
any con (11)	✓	✓	✓	✓	×	✓	✓

(1) Findler & Felleisen (2002). (2) Hinze *et al.* (2006). (3) Gronski & Flanagan (2007). (4) Blume & McAllester (2006). (5) Dimoulas *et al.* (2011). (12) Flanagan (2006). (13) Knowles & Flanagan (2010). (14) Wadler & Findler (2009). (15) Ou *et al.* (2004). (6) Does the system include dependent contracts or function types (✓) or not (×) and, for latent systems, is the semantics lax or picky? (See below for more on "indy" checking.) (7) An "unusual" form of dependency, where negative blame in the codomain results in nontermination. (17) A nondeterministic variant of CBN. (8) Do failed contracts raise labeled blame ($\Uparrow l$), raise blame without a label ($\Uparrow$), get stuck, or sometimes raise blame and sometimes diverge ($\perp$)? (9) 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 ($\bigcirc$)? (10) Is the typing relation monotonic, i.e., is the typing relation known to be uniquely defined? (11) Are arbitrary user-defined boolean functions allowed as contracts or refinements (✓), or only built-in ones (×)?

The largest difference is between latent and manifest treatments of contracts—i.e., whether contract checking (under whatever name) 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, in manifest systems, dependent function types. Latent systems with dependent contracts also vary in whether their semantics is lax or picky.

Next, most contract calculi use a standard call-by-value order of evaluation ("eval order" in the figure). Notable exceptions include those of Hinze *et al.* (2006), which is embedded in Haskell, Flanagan (2006), which uses a variant of call-by-name, and Knowles & Flanagan (2010), which uses full β-reduction (more on this 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 also 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 & 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 & Flanagan (2007) and Flanagan (2006) involve a circularity between the typing, subtyping, and implication relations. Knowles & Flanagan (2010) improve the technical presentation of λ_{H} in both respects. In particular, they avoid circularity (as we do) by introducing a denotational interpretation of types and maintain small-step evaluation by using a new syntactic form of "partially evaluated casts" (like most of the other systems).

The main contributions of the present paper are (1) the dependent 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 simple.) The nondependent part of our ϕ translation essentially coincides with the one studied by Gronski & Flanagan (2007), and our behavioral correspondence theorem is essentially the same as theirs. Our ψ translation completes their story for the nondependent case, establishing a tight connection between the systems. The full dependent forms of ϕ and ψ studied here are novel, as is the observation that the correspondence between the latent and manifest worlds is more problematic in this setting.

Our formulation of λ_{H} is most comparable to that of Knowles & Flanagan (2010), 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 significantly simplifies parts of the metatheory by allowing them to avoid introducing parallel reduction. We prefer standard call-by-value reduction because we consider blame as an exception—a computational effect—and we want to be able to reason about *which* blame will be raised by expressions involving many casts. At first glance, it might seem that our theorems follow directly from the results for Knowles and Flanagan's language, since CBV is a restriction of full β-reduction. However, 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 the term *bad* contains a cast that fails. In our system $\{y{:}B \mid \mathsf{true}\}$ is not a subtype of $\{y{:}B \mid (\lambda x{:}S.\ \mathsf{true})\ bad\}$ because the contract evaluates to blame. However, the subtyping does hold in the Knowles and Flanagan system because the predicate reduces to true.

The system studied by Ou *et al.* (2004) is also 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 Table 1—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 & Felleisen, 2002), though our presentation is slightly different in its use of checks. Hinze *et al.* (2006) adapted Findler & Felleisen-style contracts to a location-passing implementation in Haskell, using picky dependent function contracts.

Our λ_{H} type semantics in Section 4 is effectively a semantics of contracts. Blume & 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.

Dimoulas *et al.* (2011) introduce a new dialect of picky λ_{C}, where contract checks in the codomain are given a distinct negative label. If labels represent "contexts" for values, then this treats the contract as an independent context. "Indy" λ_{C} and picky λ_{C} will raise exactly the same *amount* of blame, but they will blame different labels.

Belo *et al.* (2011) at once simplify and extend the CBV λ_{H} given here. The type system is redesigned to avoid subtyping and closing substitutions, so type soundness is proved with easy syntactic methods (Wright & Felleisen, 1994). The language also allows *general refinements*—refinements of any type, not just base types—and extends the type system to polymorphism. This can be seen as completing some of the future work of Greenberg *et al.* (2010).

We have discussed only a small sample of the many papers on contracts and related ideas. We refer the reader to Knowles & Flanagan (2010) for a more comprehensive survey. Another useful resource is Wadler & Findler (2007) (technically superceded by Wadler & 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 employ various kinds of precise types, but 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 checks statically that these predicates hold.

Sage (Knowles *et al.*, 2006) and Knowles & Flanagan (2010) both support mixed static and dynamic checking of contracts, using, e.g., a theorem prover. We have not addressed this aspect of their work, since we have chosen to work directly with the core calculus λ_{H}, which for them was the target of an elaboration function.

8 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 for organizing our definitions. We used Aydemir and Weirich's LNGen tool (June 2010) for the Coq development of parallel reduction. Brian Aydemir, João Belo, Chris Casinghino, Nate Foster, and the anonymous POPL reviewers gave us helpful comments. The anonymous JFP reviewers' thorough comments significantly improved and clarified the paper. Our work has been supported by the National Science Foundation under grants 0702545 *A Practical Dependently-Typed Functional Programming Language*, 0910786 *TRELLYS*, 0534592 *Linguistic Foundations for XML View Update*, and 0915671 *Contracts for Precise Types*.

References

Ahmed, Amal. (2006). Step-indexed syntactic logical relations for recursive and quantified types. *Pages 69–83 of: European Symposium on Programming (ESOP).*

Aydemir, Brian, & Weirich, Stephanie. (June 2010). *LNgen: Tool support for locally nameless representations*. Tech. rept. MS-CIS-10-24.

Belo, João Filipe, Greenberg, Michael, Igarashi, Atsushi, & Pierce, Benjamin C. (2011). Polymorphic contracts. *Pages 18–37 of: European Symposium on Programming (ESOP).*

Blume, Matthias, & McAllester, David A. (2006). Sound and complete models of contracts. *Journal of Functional Programming (JFP)*, **16**, 375–414.

Cardelli, Luca, Martini, Simone, Mitchell, John C., & Scedrov, Andre. (1994). An extension of system F with subtyping. *Pages 4–56 of: Information and Computation*, vol. 9.

Chitil, Olaf, & Huch, Frank. (2007). Monadic, prompt lazy assertions in Haskell. *Pages 38–53 of: Asian Symposium on Programming Languages and Systems (APLAS).*

Dimoulas, Christos, Findler, Robert Bruce, Flanagan, Cormac, & Felleisen, Matthias. (2011). Correct blame for contracts: no more scapegoating. *Pages 215–226 of: Principles of Programming Languages (POPL).*

Findler, Robert Bruce, & Blume, Matthias. (2006). Contracts as pairs of projections. *Pages 226–241 of: Functional and Logic Programming (FLOPS).*

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Pages 48–59 of: International Conference on Functional Programming (ICFP).*

Flanagan, Cormac. (2006). Hybrid type checking. *Pages 245–256 of: Principles of Programming Languages (POPL).*

Greenberg, Michael, Pierce, Benjamin C., & Weirich, Stephanie. (2010). Contracts made manifest. *Pages 353–364 of: Principles of Programming Languages (POPL).*

Gronski, Jessica, & Flanagan, Cormac. (2007). Unifying hybrid types and contracts. *Pages 54–70 of: Trends in Functional Programming (TFP).*

Guha, Arjun, Matthews, Jacob, Findler, Robert Bruce, & Krishnamurthi, Shriram. (2007). Relationally-parametric polymorphic contracts. *Pages 29–40 of: Dynamic Languages Symposium (DLS).*

Hinze, Ralf, Jeuring, Johan, & Löh, Andres. (2006). Typed contracts for functional programming. *Pages 208–225 of: Functional and Logic Programming (FLOPS).*

Knowles, Kenneth, & Flanagan, Cormac. (2010). Hybrid type checking. *Acm trans. program. lang. syst. (TOPLAS)*, **32**, 6:1–6:34.

Knowles, Kenneth, Tomb, Aaron, Gronski, Jessica, Freund, Stephen N., & Flanagan, Cormac. (2006). Sage: Hybrid checking for flexible specifications. *Pages 93–104 of: Scheme and Functional Programming Workshop*.

Meyer, Bertrand. (1992). *Eiffel: the language*. Prentice-Hall, Inc.

Ou, Xinming, Tan, Gang, Mandelbaum, Yitzhak, & Walker, David. (2004). Dynamic typing with dependent types. *Pages 437–450 of: IFIP Conference on Theoretical Computer Science (TCS)*.

Pitts, A. M. (2005). Typed operational reasoning. *Chap. 7 of:* Pierce, B. C. (ed), *Advanced Topics in Types and Programming Languages*. The MIT Press.

Sewell, Peter, Nardelli, Francesco Zappa, Owens, Scott, Peskine, Gilles, Ridge, Tom, Sarkar, Susmit, & Strnisa, Rok. (2007). Ott: effective tool support for the working semanticist. *Pages 1–12 of: International Conference on Functional Programming (ICFP)*.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2008). The design and implementation of typed scheme. *Pages 395–406 of: Principles of Programming Languages (POPL)*.

Wadler, Philip, & Findler, Robert Bruce. (2007). Well-typed programs can't be blamed. *Scheme and Functional Programming Workshop*.

Wadler, Philip, & Findler, Robert Bruce. (2009). Well-typed programs can't be blamed. *Pages 1–16 of: European Symposium on Programming (ESOP)*.

Wright, Andrew K., & Felleisen, Matthias. (1994). A syntactic approach to type soundness. *Information and Computation*, **115**, 38–94.

Xu, Dana N., Peyton Jones, Simon, & Claessen, Koen. (2009). Static contract checking for Haskell. *Pages 41–52 of: Principles of Programming Languages (POPL)*.