Contracts Made Manifest

Michael Greenberg  
*University of Pennsylvania*

Benjamin C. Pierce  
*University of Pennsylvania*, bcpierce@cis.upenn.edu

Stephanie Weirich  
*University of Pennsylvania*, sweirich@cis.upenn.edu

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)

Part of the [Computer Engineering Commons](http://repository.upenn.edu/cis_papers), and the [Software Engineering Commons](http://repository.upenn.edu/cis_papers)

Recommended Citation

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich, "Contracts Made Manifest", May 2012.


© 2012 Cambridge University Press. This journal can be found online at [http://journals.cambridge.org/action/displayJournal?jid=JFP](http://journals.cambridge.org/action/displayJournal?jid=JFP)

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/751](http://repository.upenn.edu/cis_papers/751)

For more information, please contact repository@pobox.upenn.edu.
Contracts Made Manifest

Abstract
Since Findler and Felleisen (Findler, R. B. & Felleisen, M. 2002) introduced higher-order contracts, many variants have been proposed. Broadly, these fall into two groups: some follow Findler and Felleisen (2002) 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 and Flanagan (Gronski, J. & Flanagan, C. 2007), who defined a latent calculus $\lambda_C$ and a manifest calculus $\lambda_H$, gave a translation $\varphi$ from $\lambda_C$ to $\lambda_H$, and proved that if a $\lambda_C$ term reduces to a constant, so does its $\varphi$-image. We enrich their account with a translation $\psi$ from $\lambda_H$ to $\lambda_C$ 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 $\lambda_H$ and two dialects (“lax” and “picky”) of $\lambda_C$, establish type soundness—a substantial result in itself, for $\lambda_H$ — and extend $\varphi$ and $\psi$ 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.

Disciplines
Computer Engineering | Computer Sciences | Software Engineering

Comments

© 2012 Cambridge University Press. This journal can be found online at [http://journals.cambridge.org/action/displayJournal?jid=JFP](http://journals.cambridge.org/action/displayJournal?jid=JFP)

This journal article is available at ScholarlyCommons: [http://repository.upenn.edu/cis_papers/751](http://repository.upenn.edu/cis_papers/751)
Abstract

Since Findler and Felleisen (Findler, R. B. & Felleisen, M. 2002) introduced higher-order contracts, many variants have been proposed. Broadly, these fall into two groups: some follow Findler and Felleisen (2002) 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 and Flanagan (Gronski, J. & Flanagan, C. 2007), who defined a latent calculus $\lambda_C$ and a manifest calculus $\lambda_H$, gave a translation $\phi$ from $\lambda_C$ to $\lambda_H$, and proved that if a $\lambda_C$ term reduces to a constant, so does its $\phi$-image. We enrich their account with a translation $\psi$ from $\lambda_H$ to $\lambda_C$ 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 $\lambda_H$ and two dialects (“lax” and “picky”) of $\lambda_C$, establish type soundness—a substantial result in itself, for $\lambda_H$ – and extend $\phi$ and $\psi$ 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.

1 Introduction

The idea of contracts—arbitrary program predicates acting as dynamic pre- and post-conditions—was popularized by Eiffel (Meyer 1992). More recently, Findler and Felleisen (2002) introduced a $\lambda$-calculus with higher-order contracts. This calculus includes terms like $\langle\{x:\text{Int} \mid \text{pos} \; x\}\rangle_l^l$, in which a boolean predicate, pos, is applied to a run-time value $1$. This term evaluates to $1$, since pos 1 returns true. On the other hand, the term $\langle\{x:\text{Int} \mid \text{pos} \; x\}\rangle_l^l 0$ evaluates to blame, written $⇑l$, 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

\[
\langle \{x : \text{int} \mid \text{nonzero } x\} \mapsto \{x : \text{int} \mid \text{pos } x\} \rangle^{l'}
\]

"wraps" the function \( \lambda x : \text{int}. x - 1 \) 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 \( \text{pos } x \), raising \( \uparrow l \) if this fails (e.g., if the wrapped function is applied to 1).

Findler and Felleisen’s work (2002) sparked a resurgence of interest in contracts, and in the intervening years a bewildering variety of related systems has 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 (2002), Blume and McAllester (2006), Hinze et al. (2006), Chitil and Huch (2007), Guha et al. (2007), and Tobin-Hochstadt and 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 works of Ou et al. (2004), Knowles et al. (2006), and Wadler and Findler (2009).

A key feature of manifest systems is that descriptions like \( \{x : \text{int} \mid \text{nonzero } x\} \) are incorporated into the type system as refinement types. Values of refinement type are introduced via casts like \( \langle \{x : \text{int} \mid \text{true}\} \mapsto \{x : \text{int} \mid \text{nonzero } x\} \rangle^l n \), which has static type \( \{x : \text{int} \mid \text{nonzero } x\} \) and checks, dynamically, that \( n \) really is nonzero, raising \( \uparrow l \) otherwise. Similarly, \( \langle \{x : \text{int} \mid \text{nonzero } x\} \mapsto \{x : \text{int} \mid \text{pos } x\} \rangle^l n \) 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

\[
f = \langle \text{int} \mapsto \text{int} \mapsto \{x : \text{int} \mid \text{pos } x\} \mapsto \{x : \text{int} \mid \text{pos } x\} \rangle^l \ (\lambda x : \text{int}. x - 1),
\]

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

\[
f \ n \rightarrow_h \langle \text{int} \mapsto \{x : \text{int} \mid \text{pos } x\} \rangle^l \ ((\lambda x : \text{int}. x - 1) (\langle \{x : \text{int} \mid \text{pos } x\} \Rightarrow \text{int} \rangle^l n))
\]

First, \( n \) is cast from \( \{x : \text{int} \mid \text{pos } x\} \) to \( \text{int} \) (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 \( \text{int} \) to \( \{x : \text{int} \mid \text{pos } x\} \), 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 labels, while casts use one. Informally, a function contract check \( \langle c_1 \mapsto c_2 \rangle^{l'} f \)
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 $\langle R_1 \rightarrow R_2 \Rightarrow S_1 \rightarrow S_2 \rangle \uparrow l f$ 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 $\lambda H$ relate to the invariant checks of $\lambda C$? How does $\lambda H$ model $\lambda C$’s pair of polarized blame labels? These questions have 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 and Flanagan (2007) initiated a formal investigation of the connection between the latent and manifest worlds. They defined a core calculus, $\lambda C$, capturing the essence of latent contracts in a simply typed $\lambda$-calculus, and an analogous manifest calculus $\lambda H$. To compare these systems, they introduced a type-preserving translation $\phi$ from $\lambda C$ to $\lambda H$. What makes $\phi$ 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 $\phi$ preserves behavior, in the sense that if a term $t$ in $\lambda C$ evaluates to a constant $k$ or blame $\uparrow l$, then its translation $\phi(t)$ evaluates similarly.

Our work extends their work in two directions. First, we strengthen their main result by introducing a new feature-for-feature translation $\psi$ from $\lambda H$ to $\lambda C$ and proving a similar correspondence theorem for $\psi$. (We also give a new, more detailed proof of the correspondence theorem for $\phi$.) These correspondences show that the manifest and latent approaches are effectively equivalent in the non-dependent case.

Second, and more significantly, we extend the whole story to allow dependent function contracts in $\lambda C$ and dependent arrow types in $\lambda H$. 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:

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

Adding dependent contracts to $\lambda C$ 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 $\lambda C$, one that does recheck ($\text{picky } \lambda C$) and another that does not ($\text{lax } \lambda C$). 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 $\text{indy}$ 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 7.3. In $\lambda_H$, 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 definitions, plus an intricate sequence of technical lemmas involving parallel reduction to establish type soundness.

Surprisingly, the tight correspondence between $\lambda_C$ and $\lambda_H$ breaks down in the dependent case: the natural generalization of translations does not preserve behavior exactly. Indeed, we can place $\lambda_H$ between the two variants of $\lambda_C$ on an “axis of blame” (Figure 1), where evaluation behavior is preserved exactly when moving left on the axis (from picky $\lambda_C$ to $\lambda_H$ to lax $\lambda_C$), but translated terms can blame more than their pre-images when moving right.\footnote{There might, in principle, be some other way of defining $\phi$ and $\psi$ 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 $\phi$ and $\psi$ exist.} 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

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

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 $\lambda_C$) to $\lambda f : \text{Int} \to \text{Int}$. 0 and then apply the result to $\lambda x : \text{Int}$. x and 5, the final result will be 5, since $\lambda x : \text{Int}$. x does satisfy the contract $\{x : \text{Int} \mid \text{nonzero } x\} \mapsto \{y : \text{Int} \mid \text{true}\}$ and 5 satisfies the contract $\{z : \text{Int} \mid (\lambda x : \text{Int} . x) \ 0 = 0\}$. However, running the translation of $f$ in $\lambda_H$ yields an extra check, wrapping the occurrence of $f$ in the codomain contract with a cast from $\{x : \text{Int} \mid \text{nonzero } x\} \mapsto \{y : \text{Int} \mid \text{true}\}$ to $\{x : \text{Int} \mid \text{true}\} \mapsto \{y : \text{Int} \mid \text{true}\}$, 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 and Flanagan (2007), we are interested in translations that relate $\lambda_C$ and $\lambda_H$ feature for feature, i.e., mapping base contracts to base contracts and function contracts to function contracts. Translations that do not map feature for feature can give an exact treatment of blame. Consider the following dependent version of the wrap operator from Findler and Felleisen (2002). There are two cases: one for refinements of base types $B$, another for
Contracts made manifest

\[ B ::= \text{Bool} \mid \ldots \text{base types} \]
\[ k ::= \text{true} \mid \text{false} \mid \ldots \text{first-order constants} \]

Fig. 2. Base types and constants for \( \lambda_C \) and \( \lambda_H \).

dependent function contracts
\[
\begin{align*}
\phi((\{ x : B \mid t \})^{l,l'}) &= \{ x : B \mid \phi(t) \})^{l,l'} \\
\phi((x : c_1 \mapsto c_2)^{l,l'}) &= \lambda f : [x : c_1 \mapsto c_2]. \lambda x : [c_1]. \phi((c_2)^{l,l'}) (f (\phi((c_1)^{l,l'}) x))
\end{align*}
\]

We can define a similar mapping function that implements \( \lambda_H \)'s semantics as base-type contracts in lax or picky \( \lambda_C \). It is unsurprising that an exact mapping exists: \( \lambda_C \) and \( \lambda_H \) are lambda calculi that feature, among other things, a way to conditionally raise exceptions. That these languages are inter-encodable 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 that of casts relate.

In summary, our main contributions are (a) the translation \( \psi \) and a symmetric version of Gronski and Flanagan's behavioral correspondence theorem (2007), (b) the basic metatheory of (call-by-value (CBV), blame-sensitive) dependent \( \lambda_H \), (c) dependent versions of \( \phi \) and \( \psi \), and their properties with regard to \( \lambda_H \) and both dialects of \( \lambda_C \), 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 more interesting proofs.

2 The non-dependent languages

We begin in this section by defining the non-dependent versions of \( \lambda_C \) and \( \lambda_H \) and continue in Section 3 with the translations between them. The dependent languages, dependent translations, and their properties are developed in Sections 4, 6, and 7. Throughout the paper, rules prefixed with an \( E \) or an \( F \) are operational rules for \( \lambda_C \) and \( \lambda_H \), respectively. An initial \( T \) is used for \( \lambda_C \) typing rules; typing rules beginning with an \( S \) belong to \( \lambda_H \).

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

2.1 The language \( \lambda_C \)

The language \( \lambda_C \) 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
Syntax for $\lambda_C$

\[
T ::= B \mid T_1 \rightarrow T_2 \\
c ::= \{x:B \mid t\} \mid c_1 \mapsto c_2
\]

$$
\begin{align*}
t &::= x \mid k \mid \lambda x:T_1. t_2 \mid t_1 t_2 \mid \lceil l \rceil \mid \langle c \rangle^{l'} \mid \langle\{x:B \mid t_1\}, t_2, k\rangle^l \\
v &::= k \mid \lambda x:T_1. t_2 \mid \langle c \rangle^{l'} \mid \langle c_1 \mapsto c_2 \rangle^{l'} v \\
r &::= v \mid \lceil l \rceil \\
E &::= \{\} t \mid v \{\} \mid \langle\{x:B \mid t\}, \{\}, k\rangle^l
\end{align*}
$$

Evaluation contexts

Operational semantics for $\lambda_C$

$$
\begin{align*}
(\lambda x:T_1. t_2) v &\rightarrow_c t_2\{x := v\} & \text{E_BETA} \\
k v &\rightarrow_c \llbracket k \rrbracket(v) & \text{E_CONST} \\
\langle\{x:B \mid t\}\rangle^{l'} k &\rightarrow_c \langle\{x:B \mid t\}, t\{x := k\}\rangle^l & \text{E_CCHECK} \\
\langle\{x:B \mid t\}, \text{true}, k\rangle^l &\rightarrow_c k & \text{E_OK} \\
\langle\{x:B \mid t\}, \text{false}, k\rangle^l &\rightarrow_c \lceil l \rceil & \text{E_FAIL} \\
\langle c_1 \mapsto c_2 \rangle^{l'} v v' &\rightarrow_c \langle c_2 \rangle^{l''} (v (\langle c_1 \rangle^{l''} v')) & \text{E_CDECOMP} \\
E [\lceil l \rceil] &\rightarrow_c \lceil l \rceil & \text{E_BLAKE} \\
E [t_1] &\rightarrow_c E [t_2] \text{ when } t_1 \rightarrow_c t_2 & \text{E_COMPAT}
\end{align*}
$$

Typing rules for $\lambda_C$

$$
\begin{align*}
\Gamma \vdash t : T \\
T_VAR \\
\Gamma \vdash x : T \\
\Gamma \vdash k : \text{ty}_c(k) & \text{T_CONST} \\
\Gamma, x : T_1 \vdash t_2 : T_2 & \text{T_LAM} \\
\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2 & \text{T_APP} \\
\Gamma \vdash c : T & \text{T_CONTRACT} \\
\Gamma \vdash \lceil l \rceil : T & \text{T_BLAKE}
\end{align*}
$$

$$
\begin{align*}
\emptyset \vdash B & \text{ T_BASEC} \\
\{x:B \mid t_1\} &\vdash \text{true} & \text{T_FUNTC} \\
\vdash t_2 \rightarrow_c \text{true} & \text{T_CHECKING}
\end{align*}
$$

Fig. 3. Syntax and semantics for $\lambda_C$.

A contract obligation $\langle c \rangle^{l'}$ 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 $\lceil l \rceil$; on the other hand, if the context does not treat $\langle c \rangle^{l'} t$ as $c$ demands, then the negative label $l'$ will be blamed and the term will reduce to $\lceil l' \rceil$. In contexts where it is unambiguous, we refer to contract obligations simply as contracts.

The syntax and semantics of $\lambda_C$ appears in Figure 3, with some common definitions (shared with $\lambda_H$) in Figure 2. Besides the contract term $\langle c \rangle^{l'}$, $\lambda_C$ includes first-order
constants \( k \), blame, and active checks \( \{ \{ x : B \mid t_1 \}, t_2, k \} \). 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 \( \{ x : \text{Int} \mid \text{pos} \ x \} \to \{ x : \text{Int} \mid \text{nonzero} \ x \} \), but not base contracts over functions themselves, like \( \{ f : \text{Bool} \to \text{Bool} \mid f \ \text{true} = f \ \text{false} \} \).

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 constants using two constructions: the type-assignment function \( \text{ty}_c \), 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 \( K_B \)); and the denotation function \( \llbracket - \rrbracket \), which maps constants to functions from constants to constants (or blame, to allow for partiality). Denotations must agree with \( \text{ty}_c \), i.e., if \( \text{ty}_c(k) = B_1 \to B_2 \), then \( \llbracket k \rrbracket(k_1) \in K_{B_2} \) if \( k_1 \in K_{B_1} \).

The operational semantics is given in Figure 3. It includes six rules for basic (small-step, CBV) 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 \( \beta \)-reductions for abstractions. In these rules, arguments must be values \( v \). Since constants are of first order, we know that when \( E\_\text{CONST} \) reduces a well-typed application, the argument is not just a value but a constant.

The rules \( E\_\text{CCHECK}, E\_\text{OK}, E\_\text{FAIL}, \) and \( E\_\text{CDECOMP} \) describe the semantics of contracts. In \( E\_\text{CCHECK} \), 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 \( \phi \) from \( \lambda_C \) to \( \lambda_H \), since \( \lambda_H \) needs to know the target type of an active check. If the check evaluates to true, then \( E\_\text{OK} \) returns the initial constant. If false, the check has failed and a contract has been violated, so \( E\_\text{FAIL} \) steps the term to \( \uparrow l \). Higher-order contracts on a value \( v \) wait to be applied to an additional argument. That 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 \( v' \), \( E\_\text{CDECOMP} \) 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 \( \lambda_C \) (Figure 3) are mostly standard. We give types to constants using the type-assignment function \( \text{ty}_c \). Blame expressions have all types. Contracts are checked for well-formedness using the judgment \( \vdash c : T \), comprising the rules \( \text{T\_BASEC} \), which require that the checking term in a base contract returns a boolean value when supplied with a term of the right type, and \( \text{T\_FUNC} \). Note that the predicate \( t \) in a contract \( \{ x : B \mid t \} \) can contain at most \( x \) free, since we
are considering only non-dependent 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 $k$, the check expression $t_2$ has a boolean type, and the check is actually checking the right contract. The latter condition is formalized by the implication: $t_2 \rightarrow^* \text{true} \implies t_1[x := k] \rightarrow^* \text{true}$.

This requirement is needed for two reasons: first, nonsensical terms like $(\{x:\text{Int} \mid \text{pos} \ x\}, \text{true}, 0)$ should not be well typed; and second, we use this property in showing that the translations are type-preserving (see Section 6). This rule obviously makes type checking for the full “internal language” with checks undecidable, but excluding checks decidability. We could give a more precise condition—for example, that $t_1[x := k] \rightarrow^* t_2$—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.

2.2 The language $\lambda_H$

Our second core calculus, non-dependent $\lambda_H$, extends the simply typed $\lambda$-calculus with refinement types and cast expressions. The definitions appear in Figure 4. Unlike $\lambda_C$, which separates contracts from types, $\lambda_H$ combines them into refined base types $\{x:B \mid s\}$ and function types $S_1 \rightarrow S_2$. As for $\lambda_C$, 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 $\lambda_H$.) Unrefined base types $B$ are not valid types; they must be wrapped in a trivial refinement, as the raw type $\{x:B \mid \text{true}\}$.

The terms of the language are mostly standard, including variables, the same first-order constants as $\lambda_C$, blame, abstractions, and applications. The cast expression $\langle S_1 \rightarrow S_2 \Rightarrow S_1 \rightarrow S_2 \rangle$ dynamically checks that a term of type $S_1$ can be given type $S_2$. Like $\lambda_C$, active checks are used to give a small-step semantics to cast expressions.

The values of $\lambda_H$ 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 $\lambda_C$, reusing $[-]$. Type assignment is via $ty_h$, which we assume produces well-formed types (defined in Figure 4). To keep the languages in sync, we require that $ty_h$ and $ty_c$ agree on “type skeletons”: if $ty_h(k) = B_1 \rightarrow B_2$, then $ty_h(k) = \{x:B_1 \mid s\} \rightarrow \{x:B_2 \mid s\}$.

The small-step, CBV semantics in Figure 4 comprises six basic rules and two rules involving evaluation contexts $F$. Each rule corresponds closely to its $\lambda_C$ counterpart.

Notice how the decomposition rules compare. In $\lambda_C$, the term $(\langle c_1 \mapsto c_2 \rangle \langle F \ v \ v' \rangle)$ decomposes into two contract checks: $c_1$ checks the argument $v'$, and $c_2$ checks the result of the application. In $\lambda_H$ the term $(\langle S_1 \rightarrow S_2 \Rightarrow S_2 \rightarrow S_2 \rangle \langle F \ w \ w' \rangle)$ decomposes into two casts: a contravariant cast on the argument and a covariant cast on the result. The contravariant cast $(S_1 \Rightarrow S_1)$ casts $w'$ a suitable input for $w$, while $(S_1 \Rightarrow S_2)$ casts the result from $w$ applied to (the cast) $w'$. Suppose
Syntax for $\lambda_H$

$$S ::= \{x : B \mid s_1\} \mid S_1 \rightarrow S_2$$

types/contracts

$$s ::= x \mid k \mid \lambda x : S_1.s_2 \mid s_1 s_2 \mid \uparrow l \mid \langle S_1 \Rightarrow S_2 \rangle \mid \langle\{x : B \mid s_1\}, s_2, k\rangle$$

terms

$$w ::= k \mid \lambda x : S_1.s_2 \mid \langle S_1 \Rightarrow S_2 \rangle \mid \langle S_{11} \Rightarrow S_{12} \Rightarrow S_{21} \Rightarrow S_{22}\rangle \mid w$$

values

$$q ::= w \mid \uparrow l$$

results

$$F ::= \left[\right] \mid w \mid \langle\{x : B \mid s\}, [\right], k\rangle$$

evaluation contexts

Operational semantics for $\lambda_H$

$$\lambda x : S_1.s_2 \xrightarrow{\_} s_2\{x := w_2\}$$

F_BETA

$$k w \xrightarrow{\_} \{k\}_h(w)$$

F_CONST

$$\{x : B \mid s\} \Rightarrow \{x : B \mid s_2\} \xrightarrow{\_} k \xrightarrow{\_} \{x : B \mid s, s_2[x := k], k\}$$

F_OK

$$\{x : B \mid s\}, true, k\} \xrightarrow{\_} k$$

F_FAIL

$$\{x : B \mid s\}, false, k\} \xrightarrow{\_} \uparrow l$$

F_CCHECK

$$(S_{11} \Rightarrow S_{12} \Rightarrow S_{21} \Rightarrow S_{22}) \xrightarrow{\_} w \xrightarrow{\_} \uparrow l \xrightarrow{\_} \uparrow l \xrightarrow{\_} \uparrow l$$

F_CDECOMP

$$F [s_1] \xrightarrow{\_} h F [s_2]$$

F_COMPAT

Typing rules for $\lambda_H$

$$\Delta \vdash s : S$$

S_VAR

$$\Delta \vdash k : ty_h(k)$$

S_CONST

$$\vdash S_1 \quad \Delta, x : S_1 =\dashv s_2 : S_2 \quad \Delta \vdash \lambda x : S_1.s_2 : S_1 \rightarrow S_2$$

S_LAM

$$\Delta \vdash s_1 : S_1 \rightarrow S_2 \quad \Delta \vdash s_2 : S_2 \quad \Delta \vdash s : S$$

S_APP

$$\vdash S_1 \quad \vdash S_2 \quad |S_1| = |S_2| \quad \Delta \vdash \langle S_1 \Rightarrow S_2 \rangle : S_1 \rightarrow S_2$$

S_CAST

$$\frac{\Delta \vdash s_1 \quad \vdash S_2 \quad \vdash S_1 \leftarrow \vdash S_2}{\Delta \vdash s : S_1 \rightarrow \vdash S_2}$$

S_SUB

$$\emptyset \vdash k : \{x : B \mid true\} \quad \emptyset \vdash s_2 : \{x : bool \mid true\} \quad \vdash \{x : B \mid s_1\}$$

S_CHECKING

$$\vdash s_2 \rightarrow_h true \quad \text{implies} \quad S_1 \vdash \{x := k\} \rightarrow_h \{true\}$$

$$\emptyset \vdash \langle\{x : B \mid s_1\}, s_2, k\rangle : \{x : B \mid s_1\}$$

$$\vdash S_1 \leftarrow \vdash S_2$$

SSUB_REFINE

$$\forall k \in \mathcal{X}_B. (s_1 \{x := k\} \rightarrow_h \{true\} \quad \text{implies} \quad s_2 \{x := k\} \rightarrow_h \{true\})$$

$$\vdash \{x : B \mid s_1\} \leftarrow \vdash \{x : B \mid s_2\}$$

$$\vdash S_{11} \leftarrow \vdash S_{12} \quad \vdash S_{12} \leftarrow \vdash S_{21} \quad \vdash S_{21} \leftarrow \vdash S_{22}$$

SSUB_FUN

$$\frac{\vdash \{x : B \mid true\}}{\vdash \{x : B \mid true\} \vdash S_1 \leftarrow \vdash S_2}$$

SWF_RAW

$$\frac{\vdash \{x : B \mid s\}}{\vdash \{x : B \mid s\} \vdash S_1 \leftarrow \vdash S_2}$$

SWF_FUNC

$$\frac{\vdash S_1 \leftarrow \vdash S_2}{\vdash S_1 \leftarrow \vdash S_2}$$

SWF_REFINE

Fig. 4. Syntax and semantics for $\lambda_H$.
$S_{21} = \{x : \text{Int} \mid \text{pos } x\}$ and $S_{11} = \{x : B \mid \text{nonzero } x\}$. Then the check on the argument ensures that nonzero $x \rightarrow^{*} h$ true—not, as one might expect, that pos $w' \rightarrow^{*} h$ true. While it is easy to read off from a $\lambda_C$ contract exactly which checks will occur at runtime, a $\lambda_H$ 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 $\lambda_H$ (Figure 4) are also similar to those of $\lambda_C$. Just as the $\lambda_C$ rule $T$ _Contract_ checks to make sure that the contract has the right form, the $\lambda_H$ rule $S$ _Cast_ ensures that the two types in a cast are well formed and have the same simple-type skeleton, defined as $\lfloor - \rfloor : S \rightarrow T$ (pronounced “erase $S$”):

$$\lfloor \{x : B \mid s\} \rfloor = B$$

$$\lfloor S_{1} \rightarrow S_{2} \rfloor = \lfloor S_{1} \rfloor \rightarrow \lfloor S_{2} \rfloor$$

This prevents “stupid” casts, like $\langle \lceil \text{Int} \rceil \Rightarrow \lceil \text{Bool} \rceil \rangle$. We define a similar operator, $\lceil - \rceil : S \rightarrow S$ (pronounced “raw $S$”), which trivializes all refinements:

$$\lceil \{x : B \mid s\} \rceil = \{x : B \mid \text{true}\}$$

$$\lfloor S_{1} \rightarrow S_{2} \rfloor = \lfloor S_{1} \rfloor \rightarrow \lfloor S_{2} \rfloor$$

These operations apply to $\lambda_C$ contracts and types in the natural way. Type well-formedness in $\lambda_H$ is similar to contract well-formedness in $\lambda_C$, 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 $\lambda_C$, 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 $\lambda_H$ 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:

$$\langle \{x : \text{Int} \mid \text{true}\} \Rightarrow \{x : \text{Int} \mid \text{pos } x\} \rangle \quad 1 \rightarrow^{*} h \quad 1$$

The source term is well typed at $\{x : \text{Int} \mid \text{pos } x\}$. Since it evaluates to 1, we would like to have $\Delta \vdash 1 : \{x : \text{Int} \mid \text{pos } x\}$. To have type preservation in general, though, $\text{ty}_{h}(1)$ must be a subtype of $\{x : \text{Int} \mid s\}$ whenever $s\{x := 1\} \rightarrow^{*} h$ true. That is, constants of base type must have “most-specific” types. One way to satisfy this requirement is to set $\text{ty}_{h}(k) = \{x : B \mid x = k\}$ for $k \in \mathcal{X}_B$; then if $s\{x := k\} \rightarrow^{*} h$ true, we have $\vdash \text{ty}_{h}(k) <: \{x : B \mid s\}$. This approach is taken in Ou _et al._ (2004) and Knowles and Flanagan (2010).

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

3 The non-dependent translations

The latent and manifest calculi differ in a few respects. Obviously, $\lambda_C$ uses contract application and $\lambda_H$ uses casts. Second, $\lambda_C$ contracts have two labels—positive and negative—where $\lambda_H$ contracts have a single label. Finally, $\lambda_H$ has a much richer type system than $\lambda_C$. Our $\psi$ from $\lambda_H$ to $\lambda_C$ and Gronski and Flanagan’s $\phi$ (2007) from $\lambda_C$ to $\lambda_H$ 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 6; the non-dependent definitions are a straightforward restriction.

For $\psi$, translating from $\lambda_H$’s rich types to $\lambda_C$’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, $\langle \psi(S_1, S_2) \rangle^{/l}$. We define $\psi$ as two mutually recursive functions: $\psi(s)$ translates $\lambda_H$ terms to $\lambda_C$ terms; $\psi(S_1, S_2)$ translates a pair of $\lambda_H$ types—effectively, a cast—to a $\lambda_C$ contract. The latter function is defined as follows:

\[
\psi(\{ x : B \mid s_1 \}, \{ x : B \mid s_2 \}) = \{ x : B \mid \psi(s_2) \}
\]
\[
\psi(S_{11} \to S_{12}, S_{21} \to S_{22}) = \psi(S_{21}, S_{11}) \mapsto \psi(S_{12}, S_{22})
\]

We use single label on the cast in both positive and negative positions of the resulting contract, i.e.:

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

When we translate a pair of refinement types, we produce a contract that will check the predicate of the target type (like $F_{\text{CCheck}}$); when translating a pair of function types, we translate the domain contravariantly (like $F_{\text{CDecomp}}$). For example,

\[
\langle\{ x : \text{Int} | \text{nonzero} \ x \} \to \text{[Int]} \Rightarrow \text{[Int]} \Rightarrow \{ y : \text{Int} | \text{pos} \ y \} \rangle^l
\]

translates to $\langle\{ x : \text{Int} | \text{nonzero} \ x \} \mapsto \{ y : \text{Int} | \text{pos} \ y \} \rangle^{/l}$.

Translating from $\lambda_C$ to $\lambda_H$, we are moving from a simple type system to a rich one. The translation $\phi$ (essentially the same as Gronski & Flanagan’s translation (2007)) generates terms in $\lambda_H$ with raw types—$\lambda_H$ types with trivial refinements, corresponding to $\lambda_C$’s simple types. Since the translation targets raw types, the type preservation theorem is stated as “if $\Gamma \vdash t : T$ then $[\Gamma] \vdash \phi(t) : [T]$” (see Section 7.1).

Whereas the difficulty with $\psi$ is ensuring that the checks match up, the difficulty with $\phi$ is ensuring that the terms in $\lambda_C$ and $\lambda_H$ 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

\[
\phi(\{ x:B | t \}) = \{ x:B | \phi(t) \}
\]

\[
\phi(c_1 \mapsto c_2) = \phi(c_1) \rightarrow \phi(c_2)
\]

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,

\[
\langle \{ x:\text{int} | \text{nonzero } x \} \mapsto \{ y:\text{int} | \text{pos } y \} \rangle l, l'
\]

translates to the \(\lambda_H\) term

\[
\lambda f:[\text{Int} \rightarrow \text{Int}].
\]

\[
\langle \{ x:\text{int} | \text{nonzero } x \} \rightarrow \{ y:\text{int} | \text{pos } y \} \Rightarrow [\text{Int} \rightarrow \text{Int}] \rangle l' (\langle [\text{Int} \Rightarrow \{ x:\text{int} | \text{nonzero } x \} \Rightarrow \{ y:\text{int} | \text{pos } y \} \rangle l x).
\]

Unfolding the domain parts of the casts on \(f\), the domain of the negative cast ensures that \(f\)'s argument is nonzero with \(\langle [\text{Int} \Rightarrow \{ x:\text{int} | \text{nonzero } x \} \rangle l\); the domain of the positive cast does nothing, since \(\langle [\text{int} | \text{nonzero } x] \Rightarrow [\text{Int}] \rangle l\) 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 \(\lambda_H\) to keep track of blame labels, mimicking \(\lambda_C\). 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 for 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, \(\langle [x:\text{int} | \text{nonzero } x] \rangle l\) translates to

\[
\lambda x:[\text{Int}].
\]

\[
\langle [x:\text{int} | \text{nonzero } x] \Rightarrow [\text{Int}] \rangle l' (\langle [\text{Int}] \Rightarrow \{ x:\text{int} | \text{nonzero } x \} \rangle l x).
\]

Only the positive cast does anything—the negative cast into \([\text{Int}]\) always succeeds. This asymmetry is consistent with \(\lambda_C\), 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 \(\phi\) and \(\psi\) 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 \(\psi\). Interestingly, we need to set up this behavioral correspondence \textit{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 \( \lambda_C \) to dependent function contracts and \( \lambda_H \) to dependent functions. Very little needs to be changed in \( \lambda_C \), since contracts and types barely interact; the changes to \( \text{E.CDecomp} \) and \( \text{T.FunC} \) are the important ones. Adding dependency to \( \lambda_H \) is more involved. In particular, adding contexts to the subtyping judgment entails adding contexts to \( \text{SSub.Refine} \). To avoid a dangerous circularity, we define closing substitutions in terms of a separate type semantics. In addition, the new \( \text{E.CDecomp} \) rule has a slightly tricky (but necessary) asymmetry, as explained below.

4.1 Dependent \( \lambda_C \)

Dependent \( \lambda_C \) has been studied since Findler and Felleisen (2002); it received a very thorough treatment (with an untyped host language) in Blume and McAllester (2006), was ported to Haskell by Hinze et al. (2006) and Chitil and 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 and Felleisen (2002), with a few technical changes to make the proofs for \( \phi \) easier.

We have marked the changed rules with a • next to their names. The new \( \text{T.RefineC} \), \( \text{T.FunC} \), and \( \text{E.CDecomp} \) rules in Figure 5 suffice to add dependency to \( \lambda_C \). 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 \( \text{T.FunC} \) adds \( x: \langle c_1 \rangle^{l,l'} \) to the context when checking the codomain of a function contract, swapping blame labels. We add a new variable rule, \( \text{T.VarC} \), that treats \( x: \langle c \rangle \) as if it were its skeleton, \( x: [c] \). While unnecessary for \( \lambda_C \)'s metatheory, this new binding form helps \( \phi \) preserve types when translating from \( \lambda_H \) to picky \( \lambda_C \); see Section 7.1.

Two different variants of the \( \text{E.CDecomp} \) rule can be found in the literature: they are \( \text{lax} \) and \( \text{picky} \). The original rule in Findler and Felleisen (2002) is lax (like most other contract calculi): it does not recheck \( c_1 \) when substituting \( v' \) into \( c_2 \). Blume and McAllester (2006) used a picky semantics without observing their departure from Findler and Felleisen (2002); Hinze et al. (2006) choose to be picky as well, substituting \( \langle c_1 \rangle^{l,l'} \) 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 \( \lambda_C \): picky \( \lambda_C \) using only \( \text{E.CDecompPicky} \) in Sections 6.1 and 7.2, and lax \( \lambda_C \) using only \( \text{E.CDecompLax} \) in Sections 6.2 and 7.1. Accordingly, we give two sets of evaluation rules: \( \rightarrow \text{lax} \) and \( \rightarrow \text{picky} \). When we write \( \rightarrow_c \), the metavariable \( c \) ranges over picky and lax. We complete the type soundness proofs here generically, writing \( \rightarrow \), for the evaluation relation. For the translations in Section 4.2, we specify the evaluation relation that we use.

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

$$
T ::= B \mid T_1 \to T_2 \\
c ::= \{x:B \mid t\} \mid x : c_1 \mapsto c_2 \\
\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, x : \forall \eta.T
$$

Types

- $\beta$
- $\text{Contract}$
- $\text{Const}$
- $\text{CDecompLax}$
- $\text{CCheck}$
- $\text{CDecompPicky}$

Terms

- $\Gamma \vdash t_1 \to t_2$:
- $\Gamma \vdash \{x:B \mid t\}, \{\}, k$:

Evaluation contexts

- $\Gamma \vdash E [t]$:
- $\Gamma \vdash \{x:B \mid t\}$:

Operational semantics for $\lambda_c$

- $(\lambda x:T_1.t_2) v\quad \rightarrow_c\quad t_2\{x := v\}$
- $k\quad v\quad \rightarrow_c\quad \{k\}(v)$
- $\Gamma \vdash x : T\quad \rightarrow_c\quad \{\{x:B \mid t\}, t\{x := k\}, k\}$
- $\Gamma \vdash c_1 \mapsto c_2\quad \rightarrow_{\text{lax}}\quad \{c_2\{x := v'\}\} t' (v ((c_1)t' v'))$
- $\Gamma \vdash c_1 \mapsto c_2\quad \rightarrow_{\text{picky}}\quad \{c_2\{x := \langle c_1 \mapsto c_2 \rangle t' v'\}\} t' (v ((c_1)t' v'))$
- $\Gamma \vdash \{x:B \mid t\}, \text{true}, k\quad \rightarrow_c\quad k$
- $\Gamma \vdash \{x:B \mid t\}, \text{false}, k\quad \rightarrow_c\quad \emptyset$
- $\Gamma \vdash [t_1]\quad \rightarrow_c\quad E [t_2]$ when $t_1 \rightarrow_c t_2$

Contract erasure

$$
\{\{x:B \mid t\}\} = B\quad \quad [x : c_1 \mapsto c_2] = [c_1] \to [c_2]
$$

Typing rules for dependent $\lambda_c$

- $\vdash \emptyset$ T_EMPTY
- $\vdash \emptyset, x : T$ T_EXTVAR
- $\vdash \emptyset, x : T$ T_EXTVAR

- $\Gamma \vdash x : T \quad \rightarrow \quad \Gamma, x : T$ T_VAR

- $\Gamma \vdash \{x : T\} \mid t_2, \Gamma \vdash \{x : T\}$ T_LAM

- $\Gamma \vdash \{x : T\}, t_2, \Gamma \vdash \{x : T\}$ T_APP

- $\Gamma \vdash \langle e \rangle \quad \rightarrow \quad T$ T_CONTRACT

- $\vdash \emptyset, k : B \quad \rightarrow \quad \emptyset, k : B$ T_BLAME

- $\vdash \emptyset, k : B \quad \rightarrow \quad \emptyset, k : B$ T_BLAME

- $\vdash \emptyset, k : B \quad \rightarrow \quad \emptyset, k : B$ T_BLAME

- $\vdash \emptyset, k : B \quad \rightarrow \quad \emptyset, k : B$ T_BLAME

Fig. 5. Syntax and semantics for dependent $\lambda_c$. 
4.1 Theorem [Progress]: If $\emptyset \vdash t : T$ then either $t \xrightarrow{c} t'$ 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 $\xrightarrow{c}$ be either $\xrightarrow{picky}$ or $\xrightarrow{lax}$. If $t \xrightarrow{c} t'$ and $t \xrightarrow{c} t''$, then $t' = t''$.

4.3 Corollary [Coevaluation]: If $\xrightarrow{c}$ be either $\xrightarrow{picky}$ or $\xrightarrow{lax}$. $t \xrightarrow{c} r$ and $t \xrightarrow{c} r'$, then $t' \xrightarrow{c} r'$.

4.4 Lemma [Term and contract substitution]: If $\emptyset \vdash v : T'$, then
1. if $\Gamma, x : S', \Gamma' \vdash t : T$, then $\Gamma, \Gamma'\{x := v\} \vdash t\{x := v\} : T$, and
2. if $\Gamma, x : T', \Gamma' \vdash l' c : T$, then $\Gamma, \Gamma'\{x := v\} \vdash l' c\{x := v\} : T$.

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

We omit the proof for $x : c \downarrow l'$ bindings, which is similar.

4.5 Theorem [Preservation]: If $\emptyset \vdash t : T$ and $t \xrightarrow{c} t'$ then $\emptyset \vdash t' : T$.

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

4.2 Dependent $\lambda_H$

Now we come to the challenging part: Dependent $\lambda_H$ and its proof of type soundness. These results require the most complex metatheory in the paper because we need some strong properties about CBV evaluation. The full definitions are in Figures 6 and 7. As before, we have marked the changed rules with a • next to their names.

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

$$
\begin{align*}
[-] & : S \rightarrow T \\
\llbracket \{x : B \mid s\} \rrbracket &= B \\
\llbracket x : S_1 \rightarrow S_2 \rrbracket &= [S_1] \rightarrow [S_2] \\
x : S_1 \rightarrow S_2 &= 1 + |S_1| + |S_2|
\end{align*}
$$

A new dependent application rule, $S_{\text{App}}$, substitutes the argument into the result type of the application. We generalize $\text{SWF}_{\text{Refine}}$ to allow refinement-type

---

3 The benefit of a CBV semantics is a better treatment of blame. By contrast, Knowles and 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 and Flanagan will notice that our proof is significantly different from their proof. We discuss this in Section 8.
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 with the smaller type. This is similar to the function subtyping rule of \( F_\subseteq \) (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 \( \Delta \vdash x : S_{11} \rightarrow S_{12} \) and \( \Delta \vdash x : S_{21} \rightarrow S_{22} \) with equal skeletons, and values \( \Delta \vdash w : (x : S_{11} \rightarrow S_{12}) \) and \( \Delta \vdash w' : S_{21} \). Then \( \Delta \vdash ((x : S_{11} \rightarrow S_{12}) \Rightarrow x : S_{21} \rightarrow S_{22}) \Rightarrow w) \Rightarrow w' : S_{22} \{ x := w' \} \). 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 \( \Delta \vdash w ((S_{21} \Rightarrow S_{11}) \Rightarrow w') \Rightarrow S_{12} \{ x := (S_{21} \Rightarrow S_{11}) \Rightarrow w' \} \); in order to apply the codomain cast to this term, we must substitute \( \langle S_{21} \Rightarrow S_{11} \rangle \Rightarrow w' \) 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 \( \epsilon \rightarrow^* \) and steps \( \rightarrow \). 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 \( \sigma \) and reading \( \Delta \models \sigma \) as "\( \sigma \) satisfies \( \Delta \)."

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
Denotations of types

\[ s \in \llbracket \{ x : B \mid s_0 \} \rrbracket \iff s \xrightarrow{h} \lnot \lor (\exists k \in \mathcal{K}_B. s \xrightarrow{h} k \land s_0 \{ x := k \} \xrightarrow{h} \text{true}) \]

\[ s \in \llbracket x : S_1 \rightarrow S_2 \rrbracket \iff \forall q \in \llbracket S_1 \rrbracket. s q \in \llbracket S_2 \{ x := q \} \rrbracket \]

Denotations of kinds

\[ \{ x : B \mid s \} \in \llbracket \ast \rrbracket \iff \forall k \in \mathcal{K}_B. s \{ x := k \} \in \llbracket \{ x : \text{Bool} \mid \text{true} \} \rrbracket \]

\[ x : S_1 \rightarrow S_2 \in \llbracket \ast \rrbracket \iff S_1 \in \llbracket \ast \rrbracket \land \forall q \in \llbracket S_1 \rrbracket. S_2 \{ x := q \} \in \llbracket \ast \rrbracket \]

Semantic judgments

\[ \Delta \models S_1 <: S_2 \iff \forall \sigma \text{ s.t. } \Delta \models \sigma, \llbracket \sigma(S_1) \rrbracket \subseteq \llbracket \sigma(S_2) \rrbracket \]

\[ \Delta \models s : S \iff \forall \sigma \text{ s.t. } \Delta \models \sigma, \sigma(s) \in \llbracket \sigma(S) \rrbracket \]

\[ \Delta \models S \iff \forall \sigma \text{ s.t. } \Delta \models \sigma, \sigma(S) \in \llbracket \ast \rrbracket \]

Fig. 8. Type and kind semantics for dependent \( \lambda_h \).

\( \Delta \models \sigma \) in a negative position. This circularity would cause the typing rules to be non-monotonic, and so the existence of the least or the 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 other judgments. (The reader may wonder why this was not a problem in \( \lambda_c \), but notice that in \( \lambda_c \), implication is only used in \text{T.CHECKING}—which has no (real) context. If we only needed implication in the \text{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 \text{SSUB.REFINE} subtyping rule refers to \text{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 \( \lambda_h \)'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

\(^4\) Knowles and Flanagan (2010) also introduce a type semantics, but their semantics 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 their types, we must also define a denotation of kinds. Since the only kind is \( * \), its denotation \([*]\) 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 \( \{ty_h(k)\} = B \) and \( s\{x := k\} \rightarrow^*_h \text{true} \) then \( \emptyset \vdash ty_h(k) <: \{x:B \mid s\} \). 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 \[\[ ty_h(k)\]\] \); this requirement is true by our “most-specific type” assumption at base types, but must be assumed at (first-order) function types. Note that this rules out including \( \text{fix} \) 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 type semantics before proving semantic type soundness.

4.6 Lemma [Determinacy]: If \( s \rightarrow_h s' \) and \( s \rightarrow_h s'' \), then \( s' = s' \).

4.7 Corollary [Coevaluation]: If \( s \rightarrow^*_h s' \) and \( s \rightarrow^*_h q \), then \( s' \rightarrow^*_h q \).

4.8 Lemma [Expansion and contraction of \([S]\)]: If \( s \rightarrow^*_h s' \), then \( s' \in \[S\] \) if \( 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 \( \rightarrow^*_h \) 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 \( s \rightarrow^*_h q \) —i.e., either \( s \rightarrow^*_h w \) or \( s \rightarrow^*_h \uparrow l \).

Proof

By induction on \(|S|\).

\( S = \{x:B \mid s_0\} \): Suppose \( s \in \[\{x:B \mid s_0\}\] \). By definition, either \( s \rightarrow^*_h w \) or \( s \rightarrow^*_h \uparrow l \), so \( s \) normalizes.

\( S = \langle x:S_1 \rightarrow S_2 \rangle \): Suppose \( s \in \[\{x:S_1 \rightarrow S_2\}\] \). We know that for any \( q \in \[S_1]\) that \( s q \in \[S_2\{x := q\}\] \). Since \( \[S_2]\) is nonempty (by Lemma 4.10), let \( q \in \[S_1]\) . By the induction hypothesis (IH), \( s q \rightarrow^*_h w \) or \( s q \rightarrow^*_h \uparrow l \). By the definition of evaluation contexts and \( \rightarrow^*_h \), the function position is evaluated first. If the application reduces to a value (i.e., \( s q \rightarrow^*_h w \)), then first \( s q \rightarrow^*_h w q \), and so \( s \rightarrow^*_h w' \). Alternatively, the application could reduce to blame (i.e., \( s q \rightarrow^*_h \uparrow l \)). There are two ways for this to happen: either \( s \rightarrow^*_h \uparrow l \), or \( s \rightarrow^*_h w' \) and \( q \rightarrow^*_h \uparrow l \). 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 \( s \in \llbracket S \rrbracket \). However, to prove this result, we must generalize it. In the bottom of Figure 8, we define three semantic judgments 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. Proofs of (2) and (3) are in Lemma 4.21.  

The first part follows by induction on the subtyping judgment.

4.13 Lemma [Trivial refinements of constants]: If \( k \in \mathcal{K} \), then \( k \in \llbracket \{x:B \mid \text{true}\} \rrbracket \).

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.

SSUB_REFINE: 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 \( \Delta, x : \{x:B \mid \text{true}\} \vdash s_1 \Rightarrow s_2 \), which means

\[
\forall \sigma. ((\Delta, x : \{x:B \mid \text{true}\} \models \sigma \land \sigma(s_1) \rightarrow^* \text{true}) \implies \sigma(s_2) \rightarrow^* \text{true} ) \quad (*)
\]

We must show \( \Delta \models \{x:B \mid s_1\} <: \{x:B \mid s_2\} \), i.e., that \( \forall \sigma. (\Delta \models \sigma \implies \llbracket \{x:B \mid s_1\} \rrbracket \subseteq \llbracket \{x:B \mid s_2\} \rrbracket \) Let \( \sigma \) be given such that \( \Delta \models \sigma \). Suppose \( s \in \llbracket \sigma(\{x:B \mid s_1\}) \rrbracket \). By definition, either \( s \) goes to \( \mathcal{H}l \), or it goes to \( k \in \mathcal{K} \) such that \( s_1 \models x := k \rightarrow^* \text{true} \).

In the former case, \( \mathcal{H}l \in \llbracket \{x:B \mid s_2\} \rrbracket \) by definition. So consider the latter case, where \( s \rightarrow^* \mathcal{H}k \).

We already know that \( k \in \mathcal{K} \), so it remains to see that \( \sigma(s_2) \models x := k \rightarrow^* \text{true} \).

We know by assumption that \( \sigma(s_1) \models x := k \rightarrow^* \text{true} \). By Lemma 4.13, \( k \in \llbracket \{x:B \mid \text{true}\} \rrbracket \).

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

SSUB_FUN: \( \Delta \vdash (x:S_{11} \rightarrow S_{12}) <: (x:S_{21} \rightarrow S_{22}) \); 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 \( \Delta \models (x:S_{11} \rightarrow S_{12}) <: (x:S_{21} \rightarrow S_{22}) \).

Let \( \Delta \models \sigma \) and \( s \in \llbracket \sigma(x:S_{11} \rightarrow S_{12}) \rrbracket \), for some \( \sigma \). We must show, for all \( q \), that if \( q \in \llbracket \sigma(S_{21}) \rrbracket \), then \( s q \in \llbracket \sigma(S_{22}) \rrbracket \).

Let \( q \in \llbracket \sigma(S_{21}) \rrbracket \). Then \( q \in \llbracket \sigma(S_{11}) \rrbracket \). Since \( s \in \llbracket \sigma(x:S_{11} \rightarrow S_{12}) \rrbracket \), we know that \( s q \in \llbracket \sigma(S_{12}) \rrbracket \). Finally, since \( \Delta, x : S_{21} \models S_{12} <: S_{22} \) and \( \Delta, x : S_{21} \models \sigma(x := q) \), we can conclude that \( s q \in \llbracket \sigma(S_{22}) \rrbracket \), and so \( s \in \llbracket \sigma(S_{21} \rightarrow S_{22}) \rrbracket \).
The proof semantic subtype soundness 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 \( s_1 \in \llbracket x : S_1 \rightarrow S_2 \rrbracket \) and \( s_2 \in \llbracket S_1 \rrbracket \). According to S_App, the application’s type is \( S_2 \{ x := s_2 \} \). By the type semantics defined in Figure 8, if \( s_1 \in \llbracket x : S_1 \rightarrow S_2 \rrbracket \), then \( s_1 \in \llbracket S_2 \{ x := q \} \rrbracket \) for any \( q \in \llbracket S_1 \rrbracket \). Sadly, \( s_2 \) is not necessarily a result! We do know, however, that \( s_2 \in \llbracket S_1 \rrbracket \), so \( s_2 \rightarrow^* q_2 \) 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 \( \Rightarrow \) 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 CBV reduction order: the \( \beta \)-redex \( (\lambda x : S_1. s_1) s_2 \) should not be contracted unless \( s_2 \) is a

Fig. 9. Parallel reduction for dependent \( \lambda \).
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 \( \Rightarrow \) commutes with \( \Rightarrow^{*}_h \). 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.

4.15 **Lemma [Substitution of parallel-reducing terms, Lemma A3 in thy.v]:**

If \( w \Rightarrow w' \), then

1. if \( s \Rightarrow s' \) then \( s\{x := w\} \Rightarrow s'\{x := w'\} \), and
2. if \( S \Rightarrow S' \) then \( S\{x := w\} \Rightarrow S'\{x := w'\} \).

4.16 **Lemma [Parallel reduction implies co-evaluation, Lemma A20 in thy.v]:**

If \( s_1 \Rightarrow s_2 \) then \( s_1 \Rightarrow^{*}_h k \iff s_2 \Rightarrow^{*}_h k \). Similarly, \( s_1 \Rightarrow^{*}_h \#l \iff s_2 \Rightarrow^{*}_h \#l \).

An alternative strategy would be to use \( \Rightarrow \) in the typing rules and \( \Rightarrow^{*}_h \) in the operational semantics. This would simplify some of our metatheory, but it would complicate the specification of the language. Using \( \Rightarrow^{*}_h \) in the typing rules gives a clearer intuition and keeps the core system small.

4.17 **Lemma [Single parallel reduction preserves type semantics]:**

If \( S_1 \Rightarrow S_2 \) then \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket \).

**Proof**

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

4.18 **Corollary [Parallel reduction preserves type semantics]:** If \( S_1 \Rightarrow^* S_2 \) then \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket \).

4.19 **Lemma [Partial semantic substitution]:** If \( \Delta_1, x : S', \Delta_2 \vdash s : S \), and \( \Delta_1, x : S' \vdash S \), and \( \Delta_1 \vdash s' : S' \), then \( \Delta_1, \Delta_2\{x := s'\} \vdash s\{x := s'\} : S\{x := s'\} \) and \( \Delta_1, \Delta_2\{x := s'\} \vdash S\{x := s'\} \).

**Proof**

By the definition of \( \Delta \vdash \sigma \).

The semantic typing case for casts requires a separate induction.

4.20 **Lemma [Semantic typing for casts]:** If \( \Delta \vdash S_1 \) and \( \Delta \vdash S_2 \) and \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket \), then \( \Delta \vdash (S_1 \Rightarrow S_2) : x : S_1 \rightarrow S_2 \) for fresh \( x \).

**Proof**

By induction on \( |S_1| = |S_2| \), going by cases on the shape of \( S_2 \). Let \( \Delta \vdash \sigma \); we show that \( \sigma((S_1 \Rightarrow S_2)) \in \llbracket \sigma(S_1 \rightarrow S_2) \rrbracket \).

\(^5\) We conjecture that the reflexive transitive closure of a similar “CBV-respecting” variant of full \( \beta \)-reduction could be used in place of our parallel reduction. It is not clear whether it would lead to shorter proofs.
\[ S_2 = \{ x : B \mid \sigma(s_2) \} \] Let \( q \in \llbracket \sigma(S_1) \rrbracket \). If \( q = \uparrow k' \), then the applied cast goes to \( \uparrow k' \), and we are done by Lemma 4.9. So \( q = k \in \mathcal{X}_B \). By \( \text{F.CHECK} \) \( \langle S_1 \Rightarrow \{ x : B \mid \sigma(s_2) \} \rangle \) \( k \rightarrow_h \langle \{ x : B \mid \sigma(s_2) \}, \sigma(s_2) \{ x := k \}, k \rangle \). By the well-kindness of \( S_2 \), we know that \( \sigma(s_2)\{ x := k \} \in \llbracket \{ x : \text{Bool} \mid \text{true} \} \rrbracket \), 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, then that value must be \text{true} or \text{false}. If it goes to \text{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 \( \sigma(s_2)\{ x := k \} \rightarrow_h \text{true} \), so \( k \in \llbracket \sigma(\{ x : B \mid s_2 \}) \rrbracket \) by definition. Expansion (Lemma 4.8) completes the proof.

\[ S_2 = x : S_{21} \rightarrow S_{22} \]: We must have \( S_1 = x : S_{11} \rightarrow S_{12} \). Let \( q \in \llbracket \sigma(S_1) \rrbracket \); if it is blame, we are done by Lemma 4.9, so let it be a value \( w \). Let \( q' \in \llbracket \sigma(S_{21}) \rrbracket \); if it is blame we are done, so let it be a value \( w' \). By \( \text{F.CDECOMP} \):

\[
\langle \sigma(S_{12}) \{ x := \langle S_{21} \Rightarrow S_{11} \rangle \} w' \rangle \Rightarrow \sigma(S_{22}) \{ x := w' \} (w (\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11}) \rangle \) \) \]

By the IH, \( \langle \sigma(S_{21}) \Rightarrow \sigma(S_{11}) \rangle \) is semantically well-typed, so \( \langle \sigma(S_{21}) \Rightarrow \sigma(S_{11}) \rangle \) \( w' \in \llbracket \sigma(S_{11}) \rrbracket \). By strong normalization (Lemma 4.11), this term reduces (and therefore parallel reduces, by Lemma 4.4) to some \( q'' \).

We know that \( w q'' \in \llbracket \sigma(S_{12}) \{ x := q'' \} \rrbracket \) by assumption. Using parallel reduction (Corollary 4.18), we have \( \llbracket \sigma(S_{12}) \{ x := q'' \} \rrbracket = \llbracket \sigma(S_{12}) \{ x := \langle \sigma(S_{21}) \Rightarrow \sigma(S_{11}) \rangle \} \rrbracket \).

Before applying the IH, we note that \( \Delta \vdash S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle \} \) \( w' \) and \( \Delta \notvdash S_{22} \{ x := w' \} \) by Lemma 4.19. Then by the IH we see that \( \langle \sigma(S_{12}) \{ x := \langle S_{21} \Rightarrow S_{11} \rangle \} \) \( w' \) \( \Rightarrow \sigma(S_{22}) \{ x := w' \} \) \( w w'' \in \llbracket \sigma(S_{22}) \{ x := w' \} \rrbracket \).

\[ 4.21 \text{Lemma } \text{[Semantic-type soundness]}: \]

1. If \( \Delta \vdash s : S \) then \( \Delta \notvdash s : S \).
2. If \( \Delta \vdash S \) then \( \Delta \notvdash S \).

\[ \text{Proof} \]

By induction on the typing and well-formedness derivations, using Corollary 4.18 in the \( S_{\text{APP}} \) case and Lemma 4.14 in the \( S_{\text{SUB}} \) case.

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.

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

\[ \text{Proof} \]

By straightforward induction on \( s \) and \( |S| \); we reuse the (critical) context well-formedness derivation in the \( S_{\text{CHECKING}} \) case.
The substitution lemma has one complication: The operational judgment $\text{S}_{\text{IMP}}$ 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. □

## 5 The translations

We divide our treatment of translations between lax $\lambda_C$, $\lambda_H$, and picky $\lambda_C$ into two sections: one for exact translations, moving right on the axis of blame, and another for inexact translations, moving left.

**Exact translations**

```
lax $\lambda_C$ ----> $\lambda_H$ ----> picky $\lambda_C$
```

**Inexact translations, more blame in target language**

Section 6 covers the exact translations, moving left on the axis of blame from picky $\lambda_C$ to $\lambda_H$, and from $\lambda_H$ to lax $\lambda_C$. Section 7 covers the inexact translations, moving right on the axis of blame from lax $\lambda_C$ to $\lambda_H$, and from $\lambda_H$ to picky $\lambda_C$.

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 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 in Section 7 demand custom term logical relations, too. We have used σ to range over closing substitutions in \( \lambda_H \); we will use δ to range over dual closing substitutions in the logical relations.

6 Exact translations

Translations moving left on the axis of blame—from picky \( \lambda_C \) to \( \lambda_H \), and from \( \lambda_H \) to lax \( \lambda_C \)—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 \( \rightarrow_c \) in this single definition: in Section 6.1 we use this definition taking \( \rightarrow_c \) to be \( \rightarrow^{\text{picky}} \); in Section 6.2 we use this definition taking \( \rightarrow_c \) to be \( \rightarrow^{\text{lax}} \). The result correspondence is defined inductively over \( \lambda_C \)’s simple types. Blame corresponds to itself at any type. At \( B \), constants in \( \mathcal{K}_B \) 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 will define translation specific extensions of the correspondence to open terms and contracts.

6.1 Translating picky \( \lambda_C \) to \( \lambda_H \): dependent \( \phi \)

We define the full \( \phi \) for the dependent calculi in Figure 11. In the dependent case, we need to translate derivations of well-formedness and well-typing of \( \lambda_C \) contexts, terms, and contracts into \( \lambda_H \) contexts, terms, and types. We translate derivations to ensure type preservation, translating \( T_{\text{VarT}} \) and \( T_{\text{VarC}} \) 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 \( f : (x : \{x : \text{Int} \mid \text{pos } x\} \mapsto \{y : \text{Int} \mid \text{true}\}) \mapsto \{z : \text{Int} \mid f 0 = 0\} \). Note that this contract is well formed
Fun

φ

something we cannot always rely on subtyping to do, since types are not in general the codomain, so we need to restore the "variables have raw types" invariant—will not in λ depend

k

S

x

is that

λ

dependent

as raw by the subtyping relation, i.e., ∆  \vdash (∆  \vdash k : T) = k

Note that dependency is eliminated.

But the proofs are easier if we translate derivations.

\begin{align*}
\phi & : (\Gamma \vdash t : T) \rightarrow S \\
\phi(\Gamma_1, x : T, \Gamma_2 \vdash x : T) &= x \\
\phi(\Gamma_1, x : c^{l,\delta}, \Gamma_2 \vdash x : [c]) &= \langle \phi(\Gamma_1 \vdash c : c) \rangle \Rightarrow [c]^\delta x \\
\phi(\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2) &= \lambda x : [T_1]. \phi(\Gamma, x : T_1 \vdash t_2 : T_2) \\
\phi(\Gamma \vdash t_1 \vdash T_1 \rightarrow T_2) &= \phi(\Gamma \vdash t_1 : T_1 \rightarrow T_2) \phi(\Gamma \vdash t_2 : T_1) \\
\phi(\Gamma \vdash \emptyset \vdash : T_1) &= \emptyset \\
\phi(\emptyset \vdash \langle c, t, k \rangle : B) &= \langle \phi(\emptyset \vdash c : B), \phi(\emptyset \vdash t : \text{bool}), k \rangle \\
\phi(\Gamma \vdash \langle c \rangle^{l,\delta} : T) &= \lambda x : [c]. \langle \phi(\Gamma \vdash c : T) \Rightarrow [c]^\delta \Rightarrow \phi(\Gamma \vdash c : T) \rangle x
\end{align*}

where x is fresh

in λ_C, but that the 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 : \text{int} | \text{pos } x} \rightarrow [\text{int}], 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 cannot always rely on subtyping to do, since types are not in general subtypes of their raw type. By tracking the variables that were bound by contracts in λ_C, we can be sure to cast them to raw types when they are referenced. We therefore translate the contract above to f : S \rightarrow \{z : \text{int} | ((S \Rightarrow \text{int} \rightarrow \text{int})^\delta f) 0 = 0\}, where S = x : {x : \text{int} | \text{pos } x} \rightarrow [\text{int}]. This (partially) motivates the x : c^{l,\delta} binding form in dependent λ_C.

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

\begin{align*}
\{x : B \mid s\} &= \{x : B \mid \text{true}\} & [x : S_1 \rightarrow S_2] &= [S_1] \rightarrow [S_2] \\
B &= \{x : B \mid \text{true}\} & [T_1 \rightarrow T_2] &= [T_1] \rightarrow [T_2] \\
\{x : B \mid t\} &= \{x : B \mid \text{true}\} & [x : c_1 \rightarrow c_2] &= [c_1] \rightarrow [c_2]
\end{align*}

Note that dependency is eliminated.

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

\begin{align*}
\phi(x : c_1 \rightarrow c_2) &= x : \phi(c_1) \rightarrow \phi(c_2) \{x := \langle \phi(c_1) \Rightarrow [c_1]^\delta \rangle x \}
\end{align*}

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., ∆  \vdash \text{ty}_h(k) \ll [\text{ty}_c(k)]. This can be proven at base types (since we have already assumed that \text{ty}_h(k) is the “most specific type” for each k), but must be assumed for first-order constant functions. This
Contracts made manifest

\[
\begin{align*}
\text{Contract/type correspondence} \\
\{x:B \mid t \} \sim_{l'} S : T \\
\text{where } T = \lambda x.B, \Gamma \vdash t \vdash s \vdash k : \text{Bool}
\end{align*}
\]

where \( T = \lambda x.B, \Gamma \vdash t \vdash s \vdash k : \text{Bool} \).

\[
x : c_1 \mapsto c_2 \sim_{l'} x : S_1 \rightarrow S_2 : T_1 \rightarrow T_2 \iff c_1 \sim_{l'} S_1 : T_1 \land \\
\forall t \sim s : T_1, c_2[x := \langle c_1 \rangle_{l'}] t \sim_{l'} S_2[k := \langle S_1 \rangle_{l'}] s : T_2
\]

\[
\text{Dual closing substitutions} \\
\Gamma \vdash \delta \iff \begin{cases}
\forall x : T \in \Gamma. \, \delta_1(x) \sim \delta_2(x) : T \\
\forall x : c_{l'}. \in \Gamma. \, \delta_1(x) = \langle \delta_1(c) \rangle_{l'} t \land \delta_2(x) = \langle [c] \mapsto \delta_2(S) \rangle_{l'} s \\
\text{where } S = \phi(\Gamma \vdash c : [c]) \land t \sim s : [c]
\end{cases}
\]

\[
\text{Lifted to open terms} \\
\Gamma \vdash t \sim s : T \iff \forall \delta. (\Gamma \vdash \delta \text{ implies } \delta_1(t) \sim \delta_2(s) : T) \\
\Gamma \vdash c \sim_{l'} S : T \iff \forall \delta. (\Gamma \vdash \delta \text{ implies } \delta_1(c) \sim_{l'} \delta_2(S) : T)
\]

Fig. 12. Blame-exact correspondence for \( \phi \) from picky \( \lambda C \).

Slightly restricts the types we might assign to our constants, e.g., we cannot say \( \text{ty}_h(\text{sqrt}) = x : [x : \text{Float} \mid x := 0] \rightarrow \{y : \text{Float} \mid (y * y) = x\} \), since it is not the case that \( \Delta \vdash \text{ty}_h(\text{sqrt}) : [\text{Float} \rightarrow \text{Float}] \). Since its domain cannot be refined, \([\text{sqrt}]\) must be defined for all \( k \in \text{\textit{Float}} \), e.g., \([\text{sqrt}](1)\) must be defined. We have already required that denotations be total over their simple types in \( \lambda_C \), and \( \lambda_{lH} \) uses the same denotation function \([\cdot] \), so this requirement does not seem too severe. In any case, we can define it to be equal to \( \eta_0 \), for some \( \eta_0 \). We could instead translate \( k \) to \( \langle \text{ty}_h(k) \mapsto [\text{ty}_h(k)]^{\#} \rangle \), however, in this case the non-dependent 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 \( \rightarrow_{\text{picky}} \). For a binding \( x : c_{l'}. \in \Gamma \), we use \( \phi \) 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 \( x : c_{l'}. \in \Gamma \) must have come from the domain of an application of \( T_{\text{Func}} \), so the labels on the binding are \textit{already} swapped when \( \phi \) or \( \Gamma \vdash \delta \) see 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.

6.1 Lemma [Expansion and contraction]: If \( t \rightarrow_{\text{picky}}^* t' \), and \( s \rightarrow_{\#}^* s' \) then \( t \sim s : T \iff t' \sim s' : T \).

6.2 Lemma [Constants correspond to themselves]: For all \( k, k \sim k : \text{ty}_c(k) \).

6.3 Lemma [Equivalence is closed under parallel reduction]: If \( s \Rightarrow s' \) then \( t \sim s' : T \). Similarly, if \( S \Rightarrow S' \) then \( c \sim_{l'} S : T \iff c \sim_{l'} S' : T \).

\textbf{Proof}

In both cases, by induction on \( T \), using the first to prove the second. \( \square \)
6.4 Lemma [Trivial casts]: If \( t \sim s : B \) and \( \parallel S \) = \( B \), then \( t \sim \langle S \Rightarrow [B] \rangle s : B \).

6.5 Lemma [Related base casts]: If \( \{x:B \mid t\} \sim^{l_{B}} \{x:B \mid s\} : B \) and \( \parallel S \) = \( B \), then
\[ \langle \{x:B \mid t\} \rangle t' \sim \langle S \Rightarrow \{x:B \mid s\} \rangle s' : B. \]

Proof
Direct. Note that \( l_{0} \) and \( l_{1} \) are entirely irrelevant. □

6.6 Lemma [Bulletproofing]: If \( t \sim s : T \) and \( c \sim^{L} S : T \) then \( \langle c \rangle^{L} t \sim \langle S \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow s : T. \)

Proof
By induction on \( T \). First, observe that either both \( t \) and \( s \) go to \( \# \) 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 \( t \rightarrow^{picky} v, s \rightarrow^{h} w, \) and \( v \approx w : T \).

\( T = B \): So \( c = \{x:B \mid t_{1}\} \) and \( S = \{x:B \mid s_{1}\} \) and \( S' = \{x:B \mid s_{2}\} \). By Lemma 6.5 we have \( \langle c \rangle^{L} t \sim \langle S \Rightarrow \{S\} \Rightarrow s : B \). By Lemma 6.4 we can add the extra, trivial cast \( \langle S \Rightarrow [S] \rangle^{L}. \)

\( T = T_{1} \rightarrow T_{2} \): We know that \( c = x : c_{1} \rightarrow c_{2} \) and \( S = x : S_{1} \rightarrow S_{2} \). Let \( t' \sim s' : T_{1} \). We only need to consider the case where \( t' \rightarrow^{picky} v' \) and \( s' \rightarrow^{h} w' \). If \( t' \rightarrow^{picky} \# \) and \( s' \rightarrow^{h} \# \), then the outer terms correspond because both blame \( \# \).

On the \( \lambda_{C} \) side, \( \langle c \rangle^{L} t \rightarrow^{picky} \langle c_{2}\{x := \langle c_{1}\rangle^{L} v'\}\rangle^{L} (v (\langle c_{1}\rangle^{L} v')) \). In \( \lambda_{H} \), we can see
\[ \langle S \Rightarrow \{S\} \rangle^{L} \langle S \Rightarrow \{S\} \Rightarrow s \rangle s' \rightarrow^{h}. \]
\[ \langle S_{2}\{x := \langle S_{1} \Rightarrow S_{1}\rangle^{L} w\} \rangle \Rightarrow \langle S_{2}\rangle \rangle \rangle \Rightarrow \langle \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \rangle \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \rangle \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \rangle \rightarrow^{h} \]

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 6.5 and \( c_{1} \sim^{L} S_{1} : B \), we know that \( \langle c_{1}\rangle^{L} v' \sim \langle S_{1} \Rightarrow S_{1}\rangle^{L} w' : B \). Both terms go 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 \( \langle c_{1}\rangle^{L} k \rightarrow^{picky} k \) and \( \langle S_{1} \Rightarrow S_{1} \rangle^{L} k \rightarrow^{h} k \). Now the terms evaluate like so:
\[ \langle c_{2}\{x := \langle c_{1}\rangle^{L} v'\}\rangle^{L} (v (\langle c_{1}\rangle^{L} v')) \rightarrow^{picky} \langle c_{2}\{x := \langle c_{1}\rangle^{L} k\}\rangle^{L} (v k) \]
\[ \langle S_{2}\{x := \langle S_{1} \Rightarrow S_{1}\rangle^{L} k\} \Rightarrow \langle S_{2}\rangle \rangle \rangle \langle \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \rangle \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \rangle \rightarrow^{h} \]
\[ \langle S_{2}\{x := \langle S_{1} \Rightarrow S_{1}\rangle^{L} k\} \rangle \Rightarrow \langle S_{2}\rangle \rangle \rangle \langle \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \Rightarrow \{S\} \rangle \rightarrow^{h} (w (\langle S_{1} \Rightarrow [S_{1}] \rangle^{L} k)) \]

By Lemma 6.4, \( k \sim \langle S_{1} \Rightarrow [S_{1}] \rangle^{L} k : B \), so \( v k \sim w (\langle S_{1} \Rightarrow [S_{1}] \rangle^{L} k) : T_{2} \). We have by definition (and \( k \sim k : B \)) that \( c_{2}\{x := \langle c_{1}\rangle^{L} k\} \sim^{L} S_{2}\{x := \langle S_{1} \Rightarrow S_{1}\rangle^{L} k\} : T_{2} \). Recall that \( \langle [S_{1}] \Rightarrow [S_{1}] \rangle^{L} k \rightarrow^{h} k \). This implies \( \langle S_{1} \Rightarrow [S_{1}] \rangle^{L} k \Rightarrow^{h} k \) (Lemma A44 in the Coq). We can then see that
If 6.8 Lemma: to show that the implication judgment. As a preliminary, we use the behavioral correspondence Theorem 6.7.

Proof

\[ S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r k \} \Rightarrow S_2 \{ x := k \} \] by Lemma A1 in the Coq. By extension with the congruence rules:

\[
\begin{align*}
\langle S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r k \} \Rightarrow [S_2] \rangle_t^r \\
\langle \langle [S_2] \Rightarrow S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r k \} \rangle_t^r (w \ ((\langle S_1 \Rightarrow [S_1] \rangle_t^r k)) \Rightarrow \\
\langle S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r k \} \Rightarrow [S_2] \rangle_t^r \\
\langle \langle [S_2] \Rightarrow S_2 \{ x := k \} \rangle_t^r (w \ ((\langle S_1 \Rightarrow [S_1] \rangle_t^r k))
\end{align*}
\]

By the IH \( \langle c_2 \{ x := \langle c_1 \rangle_t^r k \} \rangle_t^r (v k) \) corresponds to the former, which means it is related to the latter by Lemma 6.3. We conclude the case with expansion (Lemma 6.1).

\[ T_1 = T_{11} \rightarrow T_{12} \]. We continue with an application of F_CDECOMP in \( \lambda_1 \):

\[
\begin{align*}
\langle S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r w' \} \Rightarrow [S_2] \rangle_t^r \\
((\langle [S] \Rightarrow S \rangle_t^r w) (\langle [S_1] \Rightarrow S_1 \rangle_t^r w')) \rightarrow_h \\
\langle S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r w' \} \Rightarrow [S_2] \rangle_t^r \\
\langle [S_2] \Rightarrow S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r w' \} \rangle_t^r \\
(w \ ((\langle S_1 \Rightarrow [S_1] \rangle_t^r (\langle [S_1] \Rightarrow S_1 \rangle_t^r w')))
\end{align*}
\]

By the IH on \( c_1 \sim_t^r S_1 : T_1 \) and \( v' \sim w' : T_1 \), we can find what we need for the domain: \( \langle c_1 \rangle_t^r v' \sim \langle S_1 \Rightarrow [S_1] \rangle_t^r (\langle [S_1] \Rightarrow S_1 \rangle_t^r w') : T_1 \). 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 \( c_2 \{ x := \langle c_1 \rangle_t^r v' \} \sim_t^r \langle S_2 \{ x := \langle [S_1] \Rightarrow S_1 \rangle_t^r w' \} \rangle_t^r : T_2 \) by assumption, so the IH tells us that the codomain contract and bulletproofing correspond. We conclude by expansion (Lemma 6.1). \( \square \)

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

6.7 Theorem [Behavioral correspondence]: If \( \vdash \Gamma \), then:

1. If \( \phi(\Gamma \vdash t : T) = s \) then \( \Gamma \vdash t \sim s : T \).
2. If \( \phi(\Gamma \vdash c : T) = S \) then \( \Gamma \vdash c \sim_t^r S : T \).

Proof

We simultaneously show both properties by induction on the depth of \( \phi \)'s recursion. \( \square \)

We can now prove that \( \phi \) preserves types, using Theorem 6.7 to show that \( \phi \) preserves the implication judgment. As a preliminary, we use the behavioral correspondence to show that \( \phi \) preserves the implication judgment.

6.8 Lemma: If \( t_1 \rightarrow^* \text{picky} \) true implies \( t_2 \rightarrow^* \text{picky} \) true then \( \emptyset \vdash \phi(\emptyset \vdash t_1 : \text{Bool}) \Rightarrow \phi(\emptyset \vdash t_1 : \text{Bool}) \).

Proof

By the logical relation. \( \square \)

The type preservation proof is very similar to the correspondence proof of Theorem 6.7.
Term translation
\[ \psi : s \rightarrow t \]

\[ \psi(x) = x \]
\[ \psi(\lambda x : S \cdot s) = \lambda x : [S] \cdot \psi(s) \]
\[ \psi(S_1 \Rightarrow S_2) = \psi(S_1) \Rightarrow \psi(S_2) \]
\[ \psi([\{ x : B \mid s_1 \}, s_2, k]) = \{ [x : B \mid \psi(s_1)], \psi(s_2), k \} \]

Cast translation
\[ \psi : S \times S \times l \rightarrow T \]

\[ \psi^l([\{ x : B \mid s_1 \}, \{ x : B \mid s_2 \}]) = \{ [x : B \mid \psi(s_2)] \} \]
\[ \psi^l(x : S_{11} \rightarrow S_{12}, x : S_{21} \rightarrow S_{22}) = x : \psi^l(S_{21}, S_{11}) \rightarrow \psi^l(S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle \}, S_{22}) \]

Fig. 13. \( \psi \) mapping dependent \( \lambda_H \) to dependent \( \lambda_C \).

6.9 Theorem [Type preservation]: If \( \phi(\Gamma) = \Delta \) then:

1. \( \vdash \Delta \).
2. If \( \phi(\Gamma \vdash t : T) = s \) then \( \Delta \vdash s : [T] \).
3. If \( \phi(\Gamma \vdash \lambda \cdot l' c : T) = S \) then \( \Delta \vdash S \).

Proof
We prove all three properties simultaneously, by induction on the depth of \( \phi \)'s recursion.

The proof is by cases on the \( \lambda_C \) context well-formedness/term typing/contract well-formedness derivations, which determine the branch of \( \phi \) taken.

6.2 Translating \( \lambda_H \) to lax \( \lambda_C \): dependent \( \psi \)

In this section, we formally define \( \psi \) for the dependent versions of lax \( \lambda_C \) and \( \lambda_H \).

We prove that \( \psi \) is type preserving and induces behavioral correspondence.

The full definition of \( \psi \) is given in Figure 13. Most terms are translated homomorphically. In abstractions, the annotation is translated by erasing the refined \( \lambda_H \) 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 \( \psi \) inserts new casts, we need to pick a blame label: \( \psi(\langle S_1 \Rightarrow S_2 \rangle) \) passes \( l \) as an index to \( \psi^l(S_1, S_2) \).

We use the term correspondence \( t \sim s : T \) (Figure 10), interpreting it as using \( \rightarrow_{lax} \), and define a new contract/cast correspondence \( c \sim S_1 \Rightarrow^l S_2 : T \) (Figure 14), relating contracts and pairs of \( \lambda_H \) 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 \( \psi \). Note that the correspondence is blame-exact, relating \( \lambda_C \) and \( \lambda_H \) 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.
6.12 Lemma [Constants correspond to themselves]:

For all $T$, $\{x:B \mid t\} \sim \{x:B \mid s\} \implies \{x:B \mid s_2\} : B \iff \forall k \in X_B. t[x := k] \sim s_2[x := k] : \text{Bool}$

6.10 Lemma [Expansion and contraction]:

For all $T$, $c_1 \sim c_2 \sim x:S_{11} \rightarrow S_{12} \implies S_{21} \rightarrow S_{22} : T_1 \rightarrow T_2 \iff c_1 \sim S_{12}[x := S_{11}] \implies S_{22}[x := s] : T_2$

Proof.
By induction on $T$. We reason via expansion (Lemma 6.10), showing that the initial terms reduce to corresponding terms.

6.11 Lemma [Blame corresponds to blame]:

For all $T$, $\hat{l} \sim \hat{l}' : T$.

6.12 Lemma [Constants correspond to themselves]:

For all $k$, $k \approx k : \text{ty}_c(k)$.

As a corollary of Lemmas 6.11 and 6.10, 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.

6.13 Lemma [Corresponding terms coevaluate]:

If $t \sim s : T$ then $t \longrightarrow \text{lax}^* v \land s \longrightarrow \text{lax}^* w$ or $t \longrightarrow \text{lax}^* \hat{l} \land s \longrightarrow \text{lax}^* \hat{l}'$; moreover, $t \longrightarrow \text{lax}^* r$ and $s \longrightarrow \text{lax}^* q$ such that $r \approx q : T$.

6.14 Lemma [Contract/cast correspondence]:

If $c \sim S_1 \implies S_2 : T$ and $t \sim s : T$ then $(c)_{\text{ld}} \sim \langle S_1 \implies S_2 \rangle_{\text{ld}} : s : T$.

Proof.
By induction on $T$. We reason via expansion (Lemma 6.10), showing that the initial terms reduce to corresponding terms.

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

\[
T = T_1 \rightarrow T_2 : c = x : c_1 \rightarrow c_2, S_1 = x : S_{11} \rightarrow S_{12}, \text{ and } S_2 = x : S_{21} \rightarrow S_{22}.
\]

We know by inversion of the contract/cast relation that \( c_1 \sim S_{21} \Rightarrow S_{11} : T_1 \) and that for all \( t \sim s : T_1, c_2 \{ x := t \} \sim S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle \} \Rightarrow S_{22} \{ x := s \} : T_2 \). We want to prove that \( \langle c \rangle^l \sim \langle S_1 \Rightarrow S_2 \rangle^l : T_1 \rightarrow T_2 \). First, we can assume \( t \longrightarrow_{lax}^* v \) and \( s \longrightarrow_{h}^* w \) where \( v \sim w : T_1 \rightarrow T_2 \)—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 \( \lambda_C \) side, we have

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

In \( \lambda_H \), we find

\[
\langle (S_1 \Rightarrow S_2) \rangle (s') \longrightarrow_{h}^* \langle S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle \} \Rightarrow S_{22} \{ x := w' \} \rangle^l (w \langle (S_{21} \Rightarrow S_{11} \rangle \} \Rightarrow w') : T_2.
\]

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

6.15 Lemma [Skeletal equality of subtypes]: If \( \Delta \vdash S_1 <: S_2 \), then \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket \).

6.16 Lemma: If \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket = T \), then \( \llbracket \psi^l (S_1, S_2) \rrbracket = T \).

6.17 Lemma: If \( \Delta_1 \vdash S_1 \) and \( \Delta_1 \vdash S_2 \), where \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket \) then

1. if \( \Delta_1, x : S_1, \Delta_2 \vdash s : S \) then \( \Delta_1, x : S_2, \Delta_2 \{ x := \langle S_2 \Rightarrow S_1 \rangle^l x \} \vdash \llbracket S_2 \Rightarrow S_1 \rrbracket^l \{ x := \langle S_2 \Rightarrow S_1 \rangle^l x \} \), and
2. if \( \Delta_1, x : S_1, \Delta_2 \vdash s \) then \( \Delta_1, x : S_2, \Delta_2 \{ x := \langle S_2 \Rightarrow S_1 \rangle^l x \} \vdash \llbracket S_2 \Rightarrow S_1 \rrbracket^l \{ x := \langle S_2 \Rightarrow S_1 \rangle^l x \} \).

We use the correspondence relations to show that \( s \) and its translation \( \psi(s) \) correspond—i.e., that \( \psi \) faithfully translates the \( \lambda_H \) 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 \( \psi \) does. Instead, we do induction on the depth of \( \psi \)'s recursion (and also derivation height, for the \( S_{SUB} \) case).

6.18 Theorem [Behavioral correspondence]:

1. If \( \Delta \vdash s : S \) then \( \llbracket \Delta \rrbracket \vdash \psi(s) \sim s : \llbracket S \rrbracket \).
2. If \( \Delta \vdash S_1 \) and \( \Delta \vdash S_2 \), where \( \llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket = \llbracket S \rrbracket \), then \( \llbracket \Delta \rrbracket \vdash \psi^l (S_1, S_2) \sim S_1 \Rightarrow S_2 : \llbracket S \rrbracket \) (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 \(|\Delta| : S\) (for part 1) or \(|\Delta| : S_1| + |\Delta| : S_2|\) (for part 2). The first component decreases in all uses of the IH except for the \(S_{\text{Sub}}\) 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 \(\Delta \vdash s : S\). The rule that was used determines the shape of \(\psi(s)\) in all cases but \(S_{\text{Sub}}\).

We give only the most interesting cases for the first part: \(S_{\text{Cast}}, S_{\text{Checking}},\) and \(S_{\text{Sub}}\).

\(S_{\text{Cast}}\): \(\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \rightarrow S_2\) and \(\psi((S_1 \Rightarrow S_2)^l) = (\psi^l(S_1, S_2))^l\). By inversion, \(\Delta \vdash S_1\) and \(\Delta \vdash S_2\), where \([S_1] = [S_2]\).

By the IH for proposition (2), \([\Delta] \vdash \psi^l(S_1, S_2) \sim S_1 \Rightarrow^l S_2 : [S_2]\).

Let \([\Delta] \models \delta\); we must show \(\delta_1((\psi^l(S_1, S_2))^l) \sim \delta_2((S_1 \Rightarrow S_2)^l) : [S_1] \rightarrow [S_2]\). Let \(t \sim s : [S_1]\). We have \(\delta_1((\psi^l(S_1, S_2))^l) t \sim \delta_2((S_1 \Rightarrow S_2)^l) s : [S_2]\) by Lemma 6.14.

\(S_{\text{Checking}}\): We have \(\Delta \vdash \langle \{x : B | s_1\}, s_2, k \rangle^l : \{x : B | s_1\}\); translating yields \(\psi\langle \{x : B | s_1\}, s_2, k \rangle^l = \{x : B | \psi(s_1), \psi(s_2), k\}^l\). Recall that the terms of the active check are closed. By inversion we have \(\emptyset \vdash s_2 : \{x : \text{Bool} | \text{true}\}\) and \(\emptyset \vdash k : \{x : B | \text{true}\}\), so \(k \in \mathcal{K}_B\).

By the IH, \(\psi(s_2) \sim s_2 : \text{Bool}\). 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 \(\hat{t}'\), then the checks do too, and \(\hat{t}' \approx \hat{t} : B\).
- If they go to \(\text{false}\), then the checks go to \(\hat{t}\), and \(\hat{t} \approx \hat{t} : B\).
- If they go to \(\text{true}\), then the checks go to \(k \in \mathcal{K}_B\), and \(k \approx k : B\).

\(S_{\text{Sub}}\): \(\Delta \vdash s : S\); we do not know anything about the shape of \(\psi(s)\). By inversion, \(\Delta \vdash s : S'\) and \(\Delta \vdash S' < : S\). By Lemma 6.15, \([S'] = [S]\).

Since the sub-derivation \(\Delta \vdash s : S'\) is smaller, by the IH \([\Delta] \vdash \psi(s) \sim s : [S']\). But \([S'] = [S]\), so we are done.

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

\(\psi^l(S_1, \{x : B | s_2\}) = \{x : B | \psi(s_2)\}\): Note that \(S_2 = \{x : B | s_2\}\). By inversion of \(\Delta \vdash \{x : B | s_2\}\), we have \(\Delta, x : \{x : B | \text{true}\} \vdash s_2 : \{x : \text{Bool} | \text{true}\}\).

By the IH for proposition (1), \([\Delta], x : B \vdash \psi(s_2) \sim s_2 : \text{Bool}\).

We must show \([\Delta] \vdash \{x : B | \psi(s_2)\} \sim S_1 \Rightarrow^l \{x : B | s_2\} : B\). Let \([\Delta] \models \delta\); we prove that \(\delta_1(\{x : B | \psi(s_2)\}) \sim \delta_2((S_1 \Rightarrow S_2)^l) : B\), i.e., for all \(k \in \mathcal{K}_B\), that \(\delta_1(\psi(s_2)) x := k \sim \delta_2(S_1) x := k : \text{Bool}\). Since \(k \sim k : B\), we can see this last by the IH.

\(\psi^l(x : S_{11} \rightarrow S_{12}, x : S_{21} \rightarrow S_{22}) = x : \psi^l(S_{21}, S_{11}) \rightarrow \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\}^l x\}, S_{22})\):

We can see \(S_2 = x : S_{21} \rightarrow S_{22}\) and so \(S_1 = x : S_{11} \rightarrow S_{12}\), where \([S_{21}] = [S_{11}]\) and \([S_{22}] = [S_{12}]\). By inversion, we have the following well-formedness derivations:

\[
\begin{align*}
\Delta \vdash S_{21} & \quad \Delta \vdash S_{11} \\
\Delta, x : S_{21} \vdash S_{22} & \quad \Delta, x : S_{22} \vdash S_{12}
\end{align*}
\]
We can apply the IH (contravariantly) to see
\[ [\Delta] \vdash \psi^t(S_{21}, S_{11}) \sim S_{21} \Rightarrow S_{11} : [S_{11}] \quad (*) \]

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 \([\Delta], x : [S_{21}] \vdash \psi^t(S_{21}, S_{11}) \sim S_{21} \Rightarrow S_{11} : [S_{11}] \). Now \( \Delta, x : S_{21} \vdash \langle S_{21} \Rightarrow S_{11} \rangle^t : S_{21} \rightarrow S_{11} \) and \( \Delta, x : S_{21} \vdash \langle S_{21} \Rightarrow S_{11} \rangle^t x : S_{11} \). By Lemma 6.17, we can substitute this last into \( \Delta, x : S_{11} \vdash \Delta_{12} x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \).

We apply the IH for proposition (2) on \( \Delta, x : S_{21} \vdash \Delta_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \} \) and \( \Delta, x : S_{21} \vdash \Delta_{22} \), showing
\[ [\Delta], x : [S_{21}] \vdash \psi^t(S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \}, S_{22}) \sim S_{12} x := \langle S_{21} \Rightarrow S_{11} \rangle^t S_{22} \sim S_{22} \vdash [S_{22}] \quad (**) \]

We now combine (*) and (**) to show \([\Delta] \vdash \psi^t(x : S_{11} \rightarrow S_{12}, x : S_{21} \rightarrow S_{22}) \sim x : S_{11} \rightarrow S_{12} \Rightarrow S_{21} \rightarrow S_{22} : [S_{22}] \). Let \( [\Delta] \vdash \delta \). We can apply (*) to see \( \delta_1(\psi^t(S_{11}, S_{11})) \) \( \Rightarrow \delta_2(S_{21}) \Rightarrow \delta_2(S_{11}) : [S_{11}] \). For the codomain we must show, for all \( t \sim s : [S_{11}] \), that
\[ \delta_1(\psi^t(S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \}, S_{22})) \{ x := t \} \sim \delta_2(S_{12}) \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \} \sim [S_{22}] \]

Let \( t \sim s : [S_{11}] \). Recalling that \( [S_{11}] = [S_{21}] \), observe \( [\Delta], x : [S_{21}] \vdash \delta \{ x := t, s \} \). Call this \( \delta' \). By (**) we see
\[ \delta_1(\psi^t(S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \}, S_{22})) \sim \delta_2(S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \}) \Rightarrow \delta_2(S_{22}) : [S_{22}] \]

which we can rewrite to
\[ \delta_1(\psi^t(S_{12} \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \}, S_{22})) \{ x := t \} \sim \delta_2(S_{12}) \{ x := \langle S_{21} \Rightarrow S_{11} \rangle^t x \} \sim [S_{22}] \]

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.

**6.19 Lemma:** If \( \emptyset \vdash s_1 : \{ x : \text{Bool} \mid \text{true} \} \) and \( \emptyset \vdash s_2 : \{ x : \text{Bool} \mid \text{true} \} \) and \( \emptyset \vdash s_1 \Rightarrow s_2 \), then \( \psi(s_1) \rightarrow_{\text{ lax }^*} \text{true} \) implies \( \psi(s_2) \rightarrow_{\text{ lax }^*} \text{true} \).

**Proof**

By the logical relation.

The type preservation proof is very similar to the correspondence proof of Theorem 6.18, though the function case of the type/contract correspondence is intricate.

**6.20 Theorem [Type preservation for \( \psi \):**

1. If \( \Delta \vdash s : S \) then \([\Delta] \vdash \psi(s) : [S] \).
2. If \( \Delta \vdash S_1, \Delta \vdash S_2 \), where \([S_1] = [S_2] = T \), then \([\Delta] \vdash^{\downarrow l''} \psi^t(S_1, S_2) : T \).
Proof

By induction on the lexicographically ordered pair containing (a) the depth of the recursion of the translation \( \psi \) 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 \( S,S \uparrow B \). Part 2 of the proof proceeds by case analysis on \( \psi^l(S_1, S_2) = c \).

We must have \( S_2 = x : S_21 \rightarrow S_22 \) and \( S_1 = x : S_11 \rightarrow S_12 \), where \( [S_21] = [S_11] \) and \( [S_22] = [S_12] \). By inversion, we have the following well-formedness derivations:

\[
\Delta \vdash S_21 \quad \Delta \vdash S_11 \quad \Delta, x : S_21 \vdash S_22 \quad \Delta, x : S_22 \vdash S_12
\]

By the IH \( [\Delta] \vdash^l \psi^l(S_21, S_11) : [S_11] \). Note that \( [\psi^l(S_21, S_11)] = [S_21] \).

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 \( [\Delta, x : [S_21]] \vdash^{l,l} \psi^l(S_21, S_11) : [S_11] \). Now \( \Delta, x : S_21 \vdash (S_21 \Rightarrow S_11)^l : S_21 \rightarrow S_11 \). Next \( \Delta, x : S_21 \vdash (S_21 \Rightarrow S_11)^l \rightarrow S_11. \) By Lemma 6.17, we can substitute this last into \( \Delta, x : S_11 \vdash S_12 \) and \( \Delta, x : S_21 \vdash S_12 \) for \( x : (S_21 \Rightarrow S_11)^l \rightarrow S_11 \).

By the IH for proposition (2) on \( \Delta, x : S_21 \vdash S_12 \) and \( \Delta, x : S_21 \vdash S_22 \),

\[
[\Delta, x : [S_21]] \vdash^{l,l} \psi^l(S_21, S_11)^l \vdash^{l,l} \psi^l(S_12 : (S_21 \Rightarrow S_11)^l \rightarrow S_11), S_22)] : [S_22]
\]

By Lemma 6.16, \( [\psi^l(S_21, S_11)] = [S_21] \), so we can rewrite the above derivation to

\[
[\Delta, x : [S_21]] \vdash^{l,l} \psi^l(S_21, S_11)^l \vdash^{l,l} \psi^l(S_12 : (S_21 \Rightarrow S_11)^l \rightarrow S_11), S_22)] : [S_21] \rightarrow [S_22]
\]

Now by T_FUNC

\[
[\Delta] \vdash^{l,l} x : \psi^l(S_21, S_11) \rightarrow \psi^l(S_12 : (S_21 \Rightarrow S_11)^l \rightarrow S_11), S_22)] : [S_21] \rightarrow [S_22] \quad \square
\]

7 Inexact translations

The same translations \( \phi \) and \( \psi \) 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 6, but the following two proofs use custom correspondences: one where lax \( \lambda_C \) terms correspond to \( \lambda_H \) terms (with possibly more blame), and another where \( \lambda_H \) terms correspond to picky \( \lambda_C \) terms (with possibly more blame). In both cases, the \( \lambda_C \) terms will be on the left and the \( \lambda_H \) terms on the right.

7.1 Translating lax \( \lambda_C \) to \( \lambda_H \)

Translating with \( \phi \) from terms in picky \( \lambda_C \) to exactly corresponding terms in \( \lambda_H \) was a relatively straightforward generalization of the non-dependent case; things get more interesting when we consider the translation \( \phi \) from lax \( \lambda_C \) to dependent \( \lambda_H \). We can prove that it preserves types (for terms without active checks), but we can only show a weaker behavioral correspondence: sometimes lax \( \lambda_C \) terms
terminate with values when their $\phi$-images go to blame. This weaker property is a consequence of bulletproofing, the asymmetrically substituting $F_{\text{CDecomp}}$ rule, and the extra casts inserted for type preservation (i.e., for $T_{\text{VarC}}$ derivations). This is not a weakness of our proof technique—we give a counterexample, a lax $\lambda_C$ term $\emptyset \vdash t : T$ such that $t \longrightarrow_{\text{lax}} v$ 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 $k \in K_B$. 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 7.2. We again use the term correspondence to relate contracts and $\lambda_H$ 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 $x:c \mapsto \cdash l'$ bindings, treating them as if they were $x: \lfloor c \rfloor$.

7.1 Lemma [Expansion and contraction]: If $t \longrightarrow_{\text{lax}} t'$, 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 6.11, since the correspondence here is much weaker.
7.2 Lemma [Everything corresponds to blame]: For all $t$ and $T$, $t \rightsquigarrow \uparrow l' : T$.

7.3 Lemma [Constants correspond to themselves]: For all $k$, $k \cong k : ty_c(k)$.

Proof
By induction on $ty_c(k)$, recalling that constants are of first order. □

As a corollary of Lemmas 7.2 and 7.1, if two terms evaluate to blame—or even just the $\lambda_H$ 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.

7.4 Lemma [Trivial casts]: If $t \rightsquigarrow s : B$, then $t \rightsquigarrow \langle S \Rightarrow [B] \rangle^l s : B$ for all $S$.

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

The third lemma shows that correspondence is closed under adding extra casts to the $\lambda_H$ term due to the inexactness of our behavioral correspondence. Since $\lambda_H$ terms can go to blame more often than corresponding lax $\lambda_C$ terms, we can add “extra” casts to $\lambda_H$ terms. We formalize this in the following lemma, which captures the asymmetric treatment of blame by the $\rightsquigarrow$ relation. We use it to show that the cast substituted in the codomain by $F_{\text{CDECOMP}}$ 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 $\lambda_H$ types.

7.6 Lemma [Extra casts]: If $t \rightsquigarrow s : T$ and $c_1 \rightsquigarrow S_1 : T$ and $c_2 \rightsquigarrow S_2 : T$, then $t \rightsquigarrow \langle S_1 \Rightarrow S_2 \rangle^l s : T$.

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 $S_2$.

$|S_1| = |S_2| = T$. Either $s \xrightarrow{\uparrow}^* \uparrow l'$ or $t$ and $s$ both go to corresponding values at $T$. If $s \xrightarrow{\uparrow}^* \uparrow l'$, then $\langle S_1 \Rightarrow S_2 \rangle^l s \xrightarrow{\uparrow}^* \uparrow l'$ and $t \rightsquigarrow \uparrow l' : T$ since everything is related to blame (Lemma 7.2).

Therefore, suppose that $t \xrightarrow{\text{lax}}^* v$ and $s \xrightarrow{\text{h}}^* w$ and $v \cong w : T$ in each of the following cases of induction:

$T = B$: So $S_2 = \{x : B \mid s_2\}$, and $c_2 = \{x : B \mid t_2\}$. $S_1 \Rightarrow S_2$ for $k \in \mathcal{N}_B$. If $t$ and $s$ both go to $k$, then $\langle S_1 \Rightarrow S_2 \rangle^l s \xrightarrow{\text{h}}^* \langle\{x : B \mid s_2\}, s_2\{x := k\}, k\rangle^l$. By $c_2 \cong S_2 : B$ we see (in particular) $t_2\{x := k\} \rightsquigarrow s_2\{x := k\} : \text{Bool}$. So $s_2\{x := k\}$ either goes to $\uparrow l'$ or $s_2\{x := k\}$ (and, irrelevantly, $t_2\{x := k\}$) go to some $k' \in \mathcal{N}_\text{Bool}$. In the former case, $\langle\{x : B \mid s_2\}, s_2\{x := k\}, k\rangle^l \xrightarrow{\text{h}}^* \uparrow l'$ and we are done (by Lemma 7.2). In the latter case, the $\lambda_H$ term either goes to $\uparrow l$ (and everything is related to blame) or goes to $k$—but so does $t$, and $k \equiv k : B$. \[\square\]
\[ T = T_1 → T_2: \] We have:

\[
S_1 = x : S_{11} → S_{12} \quad S_2 = x : S_{21} → S_{22} \\
c_1 = x : c_{11} ↦ c_{12} \quad c_2 = x : c_{21} ↦ c_{22} \\
\]

We have \( t →_{\text{Iax}}^* v \) and \( s →_{\ast}^* w \), where \( v \succsim w : T_1 → T_2 \).

Let \( t' \sim_\succ s' : T_1 \); we wish to see that \( v t' \sim_\succ (⟨S_1 ⇒ S_2⟩^l w) s' : T_2 \). Either \( s' →_{\ast}^* \hat{l}' \) or both go to values. In the former case the whole cast goes to \( \hat{l}' \) we are done by Lemma 7.2, so let \( t' →_{\text{Iax}}^* v' \) and \( s' →_{\ast}^* w' \).

Decomposing the cast in \( \hat{\lambda}_H \),

\[
(⟨S_1 ⇒ S_2⟩^l w) s' →_{\ast}^* \\
⟨S_{12}\{x := ⟨S_{21} ⇒ S_{11}⟩^l w'⟩⟩ S_{22}\{x := w'⟩⟩ (w (⟨S_{21} ⇒ S_{11}⟩^l w'))
\]

We have \( c_{21} \sim_\succ S_{21} : T_1 \) and \( c_{11} \sim_\succ S_{11} : T_1 \), so \( v' \sim_\succ ⟨S_{21} ⇒ S_{11}⟩^l w' : T_1 \) by the IH. Since \( v \succsim w : T_1 → T_2 \), we can see that \( v v' \sim_\succ w (⟨S_{21} ⇒ S_{11}⟩^l w') : T_2 \).

Furthermore, we know that for all \( t'' \sim_\succ s'' : T_1 \) that

- \( c_{12}\{x := t''⟩⟩ S_{12}\{x := s''⟩⟩ : T_2 \) and
- \( c_{22}\{x := t''⟩⟩ S_{22}\{x := s''⟩⟩ : T_2 \).

We know that \( v' \sim_\succ w' : T_1 \) and \( v' \sim_\succ ⟨S_{21} ⇒ S_{11}⟩^l w' : T_1 \), so we can see

- \( c_{12}\{x := v''⟩⟩ S_{12}\{x := w''⟩⟩ : T_2 \) and
- \( c_{22}\{x := v''⟩⟩ S_{22}\{x := ⟨S_{21} ⇒ S_{11}⟩^l w''⟩⟩ : T_2 \).

So by the IH,

\[
v v' \sim_\succ ⟨S_{12}\{x := ⟨S_{21} ⇒ S_{11}⟩^l w'⟩⟩ S_{22}\{x := w'⟩⟩ (w (⟨S_{21} ⇒ S_{11}⟩^l w')) : T_2
\]

and we are done by expansion (Lemma 7.1). □

To apply the extra cast lemma, we will need these “witness” contracts for raw types; to that end we define trivial contracts. These contracts are \( \hat{\lambda}_C \) correlate to \( \hat{\lambda}_H \)'s raw types.

\[
B \uparrow = \{ x : B \mid \text{true} \} \\
(T_1 \rightarrow T_2) \uparrow = (T_1 \uparrow) \rightarrow (T_2 \uparrow)
\]

7.7 Lemma [Lifted types logically relate to raw types]: For all \( T, T \uparrow \sim_\succ [T] : T \).

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 \( T = B → T_2 \).

7.8 Lemma [Bulletproofing]: If \( t \sim_\succ s : T \) and \( c \sim_\succ S : T \) and \( c \sim_\succ S' : T \), then \( ⟨c⟩^{l'} \ t \sim_\succ [S']^{l'} ⟨[S] \Rightarrow S⟩^{l'} s : T \).

Proof

By induction on \( T \). First, observe that either \( s →_{\ast}^* \hat{l}'' \) or both \( t \) and \( s \) go to values related at \( T \). In the former case, \( ⟨S' ⇒ [S']⟩^{l'} ⟨[S] \Rightarrow S⟩^{l'} s →_{\ast}^* \hat{l}'' \), and everything is related to blame (Lemma 7.2). So \( t →_{\text{Iax}}^* v, s →_{\ast}^* w, \) and \( v \succsim w : T \).
\(T = B\): So \(c = \{x : B | t_1\}\) and \(S = \{x : B | s_1\}\) and \(S' = \{x : B | s_2\}\). By Lemma 7.5 we have \(\langle c \rangle^{l, t} \sim s : B\). By Lemma 7.4 we can add the extra, trivial cast.

\(T = T_1 \rightarrow T_2\): We know that \(c = x : c_1 \rightarrow c_2, S = x : S_1 \rightarrow S_2\) and \(S' = x : S'_1 \rightarrow S'_2\). Let \(t' \sim s' : T_1\). By Lemma 7.2, we only need to consider the case where \(t' \rightarrow^{lax} v'\) and \(s' \rightarrow^* w'\). If \(s' \rightarrow^* \# l''\) we are done.

On the \(\lambda_C\) side, \(\langle c \rangle^{l, t} t' \rightarrow^{lax} \langle c_2(x := v') \rangle^{l, t'} (v ((c_1)_{l, d} v'))\). In \(\lambda_H\), we can see

\[\langle S' \Rightarrow \langle [S] \Rightarrow [S] \rangle \rangle (S) s' \rightarrow^*\]
\[\langle S'_2 \rangle \Rightarrow \langle [S_1] \Rightarrow [S'_1] \rangle \Rightarrow [S'_2] \Rightarrow \langle [S'] \Rightarrow [S] \rangle \rangle (\langle [S_1] \Rightarrow [S] \rangle (\langle [S_1] \Rightarrow [S_1] \rangle) (v ((c_1)_{l, d} v'))\)

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 7.5 and \(c_1 \sim \langle [S'_1] \Rightarrow [S'_1] \rangle w' : B\). The \(\lambda_H\) term goes to blame or both terms go to the same value, \(v' = w' = k \in \mathcal{K}_B\). In the former case, the entire \(\lambda_H\) term goes to blame and we are done by Lemma 7.2. So suppose \(\langle c_1 \rangle^{l, d} k \rightarrow^{lax} k\) and \(\langle [S_1] \Rightarrow [S'_1] \rangle w' \rightarrow^* k\). Now the terms evaluate like so:

\[\langle c_2(x := v') \rangle^{l, t'} (v ((c_1)_{l, d} v')) \rightarrow^{lax} \langle c_2(x := k) \rangle^{l, t'} (v k)\]

By Lemma 7.4, \(k \sim \langle [S_1] \Rightarrow [S'_1] \rangle k : B\), so \(v k \sim w ((S_1 \Rightarrow [S_1]) k) : T_2\).

Noting that \(k \sim k : B\) and \(k \sim \langle [S_1] \Rightarrow [S'_1] \rangle k : B\), we can see that \(c_2(x := k) \sim S_2(x := k) : T_2\) and \(c_2(x := k) \sim S'_2(x := (S_1 \Rightarrow [S_1]) k) : T_2\). Now the IH shows that \(\langle c_2(x := k) \rangle^{l, d} ((v k) \sim \langle S'_2(x := [S'_1] \Rightarrow [S'_1] w') = [S'_2] \rangle \langle [S_2] \Rightarrow S_2(x := k) \rangle (w ((S_1 \Rightarrow [S_1]) k)) : T_2\), and we conclude the case with expansion (Lemma 7.1).

\(- T_1 = T_1 \rightarrow T_12\). We can continue with an application of \(F.\text{CDECOMP}\) in \(\lambda_H\) and find

\[\langle S'_2 \rangle \Rightarrow \langle [S'_1] \Rightarrow [S'_1] \rangle \Rightarrow [S'_2] \rangle \Rightarrow \langle [S'] \Rightarrow [S] \rangle \rangle (w ((S_1 \Rightarrow [S_1]) (\langle [S'_1] \Rightarrow [S'_1] w')\)) \rightarrow^*\]

By the IH, \(\langle c_1 \rangle^{l, d} v' \sim \langle [S_1] \Rightarrow [S'_1] \rangle (\langle [S'_1] \Rightarrow [S'_1] w') \rightarrow^* T_1\). 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 $c_1 \leadsto S_1' : T_1$, and by Lemma 7.7 $T_1 \uparrow \leadsto [S_1'] : T_1$. Since $v' \leadsto w' : T_1$, Lemma 6.6 shows $v' \leadsto \langle [S_1'] \Rightarrow S_1' \rangle w' : T_1$. This lets us see that $c_2 \{x := v'\} \leadsto S_2' \{x := \langle [S_1'] \Rightarrow S_1' \rangle l' \} : T_2$. Now the IH and expansion (Lemma 7.1) complete the proof.

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

**7.9 Theorem [Behavioral correspondence]:** If $\Gamma \vdash t : T$, then

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

**Proof**

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

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

We find a weak corollary: $\phi(\Gamma \vdash t : B) \longrightarrow^* k$ implies $t \longrightarrow^{lax*} k$ : if the $\lambda_H$ term does not go to blame, then the original $\lambda_C$ 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 7.9 is not strong enough to preserve the implication judgment. We only expect these checks to occur at runtime, so this is good enough: $\phi$ preserves the types of source programs.)

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

1. $\vdash \Delta$.
2. $\Delta \vdash \phi(\Gamma \vdash t : T) : [T]$.
3. $\Delta \vdash \phi(\Gamma \vdash l,l' c : T)$.

**Proof**

We prove all three properties simultaneously, by induction on the depth of $\phi$’s recursion.

The proof is by cases on the $\lambda_C$ context well-formedness/term typing/contract well-formedness derivations, which determine the branch of $\phi$ taken.

To see that $\phi$ in Figure 11 does not give us exact blame, let us look at two counterexamples; in both cases, a lax $\lambda_C$ term goes to a value while its translation goes to blame. In the first example, blame is raised in $\lambda_H$ 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

\[
\begin{align*}
c &= f : (x : \{x : \text{Int} \mid \text{true}\} \mapsto \{y : \text{Int} \mid \text{nonzero} \; y\}) \mapsto \{z : \text{Int} \mid f \; 0 = 0\} \\
S_1 &= x : \{x : \text{Int} \mid \text{true}\} \mapsto \{y : \text{Int} \mid \text{nonzero} \; y\} \\
S &= \phi(0 \vdash l c : (\text{Int} \mapsto \text{Int}) \mapsto \text{Int}) \\
&= f : S_1 \mapsto \{z : \text{Int} \mid (\langle S_1 \Rightarrow [S_1]\rangle l f) \; 0 = 0\}.
\end{align*}
\]
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 $\langle c \rangle^{l_d} (\lambda f.0) (\lambda x.0) \rightarrow_{lax}^* 0$ but

$$(\lambda x : [c] \cdot (\langle S \Rightarrow [S] \rangle^{l_d} \cdot (\langle [S] \Rightarrow [S] \rangle^{l_d} x))) (\lambda f.0) (\lambda x.0) \rightarrow_{h}^* \emptyset l.$$  

For the second counterexample, let

$$c' = f : (x : \{x : \text{Int} | \text{nonzero } x\} \mapsto \{y : \text{Int} | f \ 0 = 0\})$$

$$S'_1 = x : \{x : \text{Int} | \text{nonzero } x\} \rightarrow \{y : \text{Int} | \text{true}\}$$

$$S' = \phi(\emptyset \vdash \emptyset l, l_c : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int})$$

$$= f : S'_1 \rightarrow \{z : \text{Int} | (\langle S'_1 \Rightarrow [S'_1] \rangle^{l_d} f) \ 0 = 0\}.$$  

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

$$(\lambda x : [c'] \cdot (\langle S' \Rightarrow [S'] \rangle^{l_d} \cdot (\langle S' \Rightarrow [S'] \rangle^{l_d} x))) (\lambda f.0) (\lambda x.0) \rightarrow_{h}^* \emptyset l.$$  

The extra casts that $\phi$ inserts are all necessary—none can be removed. So while variations on this $\phi$ are possible, they can only add more casts, which won't resolve the problem that $\lambda_H$ blames more.

### 7.2 Translating $\lambda_H$ to picky $\lambda_C$

Terms in $\lambda_H$ and their $\psi$-images in lax $\lambda_C$ correspond exactly, as shown in Section 6.2. When we change the operational semantics of $\lambda_C$ to be picky, however, $\psi(s)$ blames (strictly) more often than $s$. Nevertheless, we can show an inexact correspondence, as we did for $\phi$ and lax $\lambda_C$ in Section 7.1. We use a logical relation $[\sim_{\prec}]$ for $[\psi]$ into picky $\lambda_C$ (Figure 16). Here we have reversed the asymmetry: picky $\lambda_C$ may blame more than $\lambda_H$. 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).

**7.11 Lemma [Expansion and contraction]:** If $t \rightarrow_{picky}^* t'$ and $s \rightarrow_{h}^* s'$ then $t \prec_{\prec} s : T$ iff $t' \sim_{\prec} s' : T$.

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

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

**Proof**

By induction on $ty_c(k)$, recalling that constants are of first order.  

As a corollary of Lemmas 7.12 and 7.11, if a picky $\lambda_C$ term evaluates to blame, then it corresponds to any $\lambda_H$ term. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

**7.14 Lemma [Extra contracts]:** If $t \prec_{\prec} s : T$ and $c \prec_{\prec} S_1 \Rightarrow S_2 : T$ then $\langle c \rangle^{l_d} t \prec_{\prec} s : T$.  

Value correspondence

\[ \forall w : T \quad v \approx_S w \iff k \in \mathcal{K}_B \]

\[ k \approx_S t : B \iff k \in \mathcal{K}_B \]

Term correspondence

\[ t \approx_S s : T \iff t \longrightarrow^{picky^*} \| l \lor t \longrightarrow^{picky^*} v \land s \longrightarrow^*_{\mathcal{K}} w \land v \approx_S w : T \]

Contract/type correspondence

\[ \{x : B \mid t\} \approx_S \{x : B \mid s_1\} \Rightarrow \{x : B \mid s_2\} : B \iff \forall k \in \mathcal{K}_B, t \{x := k\} \approx_S \{x := k\} : \text{Bool} \]

\[ x : c_1 \rightarrow c_2 \approx_S x : S_1 \rightarrow S_2 \Rightarrow x : S_1 \rightarrow S_2 \rightarrow t : T_2 \iff c_1 \approx_S S_{11} = S_{12} = T_1 \land \forall l. \forall t \approx_S s : T_1, c_2 \{x := t\} \approx_S S_{21} \{x := \langle S_{21} \Rightarrow S_{11}\rangle\} \Rightarrow S_{22} \{x := s\} : T_2 \]

Dual closing substitutions

\[ \Gamma \vdash \delta \iff \forall x \in \text{dom}(\Gamma), \delta_1(x) \sim_S \delta_2(x) : [\Gamma(x)] \]

Lifted to open terms

\[ \Gamma \vdash t \sim_S s : T \iff \forall \delta. (\Gamma \vdash \delta \implies \delta_1(t) \sim_S \delta_2(s) : T) \]

\[ \Gamma \vdash c \sim_S S_1 \Rightarrow S_2 : T \iff \forall \delta. (\Gamma \vdash \delta \implies \delta_1(c) \sim_S \delta_2(S_1) \Rightarrow \delta_2(S) : T) \]

Fig. 16. Blame-inexact correspondence for \( \psi \) into picky \( \lambda_C \).

Proof

By induction on \( T \). If \( t \longrightarrow^{picky^*} \| l'' \) we are done, so let \( t \longrightarrow^{picky^*} v \) and \( s \longrightarrow^*_{\mathcal{K}} w \) such that \( v \approx_S w : T \).

\[ T = B : \text{So } c = \{x : B \mid t_2\} \text{ and } S_2 = \{x : B \mid s_2\}. \text{ Moreover, } v = w = k \in \mathcal{K}_B, \text{ since those are the only corresponding values at } B. \]

We can step and see \( \langle c \rangle^{l''} t \longrightarrow^{picky^*} \langle c, t_2 \{x := k\}, k \rangle' \). We know that \( t_2 \{x := k\} \sim_S s_2 \{x := k\} : \text{Bool} \). There are two possibilities: either \( t_2 \{x := k\} \longrightarrow^{picky^*} \| l'' \) or both terms go to corresponding Booleans. In the former case, the whole \( \lambda_C \) term goes to blame and we are done by Lemma 7.12. If both go to false, then the outer \( \lambda_C \) term evaluates to \( \| l \) and we are done by Lemma 7.12 again. If both go to true, then both outer terms go to \( k \), and \( k \approx_S k : B \).

\[ T = T_1 \rightarrow T_2 : \text{So } c = x : c_1 \rightarrow c_2 \text{ and } S_1 = x : S_1 \rightarrow S_1 \text{ and } S_2 = x : S_2 \rightarrow S_2. \text{ Let } t' \sim_S s' : T_1. \text{ If } t' \longrightarrow^{picky^*} l'' \text{ we are done by Lemm 7.12, so let } t' \longrightarrow^{picky^*} v' \text{ and } s' \longrightarrow^*_{\mathcal{K}} w', \text{ where } v' \approx_S w' : T_1. \text{ We want to prove } ((\langle c \rangle^{l''} t) \ t') \sim_S s' \ s' : T_2, \text{ which is true iff:} \]

\[ \langle c_2 \{x := (c_1)^{l'} v'\} \rangle^{l''} (v (\langle c_1 \rangle^{l'} v')) \sim_S w' : T_2 \]

By the IH on \( v \sim_S w' : T_1 \) and \( c_1 \sim_S S_{21} \Rightarrow S_{11} : T_1 \), we have \( \langle c_1 \rangle^{l'} v' \sim_S w' : T_1. \text{ By definition, applying } v \text{ and } w \text{ yields related terms at } T_2. \text{ Since } \langle c_1 \rangle^{l'} v' \sim_S w' : T_1, \text{ we have } c_2 \{x := c_1 \} \sim_S S_{21} \{x := S_{21} \Rightarrow \} \]
Proof

By induction on $T$. We reason via expansion (Lemma 7.11), showing that the initial terms reduce to corresponding terms.

**Case $T = B$:** So $c = \{x:B \mid t_1\}, S_1 = \{x:B \mid s_1\}$, and $S_2 = \{x:B \mid s_2\}$. Since $t \sim_s s : B$, we know that they either both reduce to $k \in \mathcal{X}_B$ or $t \longrightarrow^{\mathrm{picky}} \downarrow l'$. If the latter is the case, we are done. So suppose $t \longrightarrow^{\mathrm{picky}} k$ along with $s \longrightarrow^h k$.

We can step our terms into active checks as follows:

$$\langle x:B \mid t_1 \rangle^l \longrightarrow^{\mathrm{picky}}^* \langle x:B \mid t_1, t \rangle^l \langle x := k, k \rangle^l$$

$$\langle x:B \mid s \rangle^l \longrightarrow^h \langle x:B \mid s_1 \rangle$$

$$\langle x:B \mid t \rangle^l \longrightarrow^{\mathrm{picky}}^* \langle x:B \mid t_1 \rangle^l \langle x := k, k \rangle^l$$

By the contract/cast correspondence, we know that $t_1|x := k\rangle \sim_s s_2\{x := k\} : \text{Bool}$, so either $t_1|x := k\rangle$ goes to blame or both terms go to a $\text{Bool}$ together. In the former case, the outer $\lambda_C$ term goes to blame and we are done by Lemma 7.12. If they go to false, then both the active check goes to $\downarrow l'$ and we are done, again by Lemma 7.12. Finally, if they both go to true, then both terms will evaluate to $k \in \mathcal{X}_B$, and $k \approx_s k : B$.

**Case $T = T_1 \leftarrow T_2$:** $c = \{x : c_1 \leftarrow c_2, S_1 = x : S_{11} \rightarrow S_{12}, S_2 = x : S_{21} \rightarrow S_{22}\}$. We know by inversion of the contract/cast relation that $c_1 \sim_s S_{12} \Rightarrow S_{11} : T_1$ and that for all $l''$ and $t \sim_s s : T_1$, $c_2\{x := t\} \sim_s S_{12}\{x := S_{21} \Rightarrow S_{11} \}^{l''} s \Rightarrow S_{22}\{x := s\} : T_2$. We want to prove that $\langle c \rangle^{l,l'} \sim_s \langle S_1 \Rightarrow S_2 \rangle^{l'} s : T_1 \rightarrow T_2$. First, we can assume $t \longrightarrow^{\mathrm{picky}}^* v$ and $s \longrightarrow^h w$, where $v \sim_s w : T_1 \rightarrow T_2$—if not, both the contracted terms go to blame and we are done by Lemma 7.12.

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

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

In $\lambda_{\text{IH}}$, we find

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

By the IH, we know that $\langle c_1 \rangle^{l,l'} v' \sim_s \langle S_{21} \Rightarrow S_{11} \}^{l''} w' : T_1$. Since $v \sim_s w : T_1 \rightarrow T_2$, we have $v (\langle c_1 \rangle^{l,l'} v') \sim_s w (\langle S_{21} \Rightarrow S_{11} \}^{l''} w') : T_2$. By Lemma 7.14, $\langle c_1 \rangle^{l,l'} v' \sim_s w' : T_1$. We can then see that $c_2\{x := \langle c_1 \rangle^{l,l'} v'\} \sim_s S_{12}\{x := S_{21} \Rightarrow S_{11} \}^{l''} w' \Rightarrow S_{22}\{x := w'\} : T_2$. By the IH, we therefore have

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

$$\langle S_{12}\{x := S_{21} \Rightarrow S_{11} \}^{l''} w' \rangle \Rightarrow S_{22}\{x := w'\}^{l''} (w (\langle S_{21} \Rightarrow S_{11} \}^{l''} w')) : T_2 \square$$
7.16 Theorem [Behavioral correspondence]:

1. If $\Delta \vdash s : S$ then $[\Delta] \vdash \psi(s) \sim s : [S]$.
2. If $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $[S_1] = [S_2] = [S]$, then $[\Delta] \vdash \psi'(S_1, S_2) \sim S_1 \Rightarrow S_2 : [S]$.

Proof
By an induction similar to the proof of Theorem 6.18. □

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

1. If $\Delta \vdash s : S$ then $[\Delta] \vdash \psi(s) : [S]$.
2. If $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $[S_1] = [S_2] = T$, then $[\Delta] \vdash^T \psi'(S_1, S_2) : T$.

Proof
By an induction similar to the proof of Theorem 6.20. □

Here is an example where a $\lambda_H$ term reduces to a value while its $\psi$-image in picky $\lambda_C$ 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 nonzero value, but the codomain requires that it returns 0.

$$S_1 = f : S_{11} \rightarrow S_{12}$$
$$= f : (x : \text{Int} \rightarrow \{y : \text{Int} \mid \text{nonzero } y\}) \rightarrow \text{Int}$$
$$S_2 = f : S_{21} \rightarrow S_{22}$$
$$= f : (x : \text{Int} \rightarrow \text{Int}) \rightarrow \{z : \text{Int} \mid f \ 0 = 0\}$$
$$c = \psi'(S_1, S_2)$$
$$= f : \psi'(S_{21}, S_{11}) \mapsto \psi'(S_{12} \{f := \langle S_{21} \Rightarrow S_{11} \rangle \ f, S_{22}\})$$
$$= f : (x : \{x : \text{Int} \mid \text{true}\} \mapsto \{y : \text{Int} \mid \text{nonzero } y\}) \mapsto \{z : \text{Int} \mid f \ 0 = 0\}$$

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

$$\psi((S_1 \Rightarrow S_{21})^I w) w') = ((\psi'(S_1, S_2))^{\langle I \rangle} \psi(w)) \psi(w') = ((c)^{\langle I \rangle} \lambda \text{x : Int}. 0) \lambda \text{x : Int}. 0.$$

On the one hand, $(S_1 \Rightarrow S_{21})^I w \rightarrow^* 0$, while $(c)^{\langle I \rangle} \lambda \text{x : Int}. 0) \lambda \text{x : Int}. 0 \rightarrow^\text{picky} \emptyset$. This means we cannot hope to use $\psi$ as an exact correspondence between $\lambda_H$ and picky $\lambda_C$. (Removing the extra cast $\psi$ inserts into $S_{12}$ does not affect our example, since $\psi$ ignores $S_{12}$ here.) For example,

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

7.3 Alternative calculi

There are three alternative calculi that we have not considered here: indy $\lambda_C$ (Dimoulas et al. 2011), superpicky $\lambda_H$, 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 $\lambda_C$, representing the contract itself; we write it here as a subscript. They accordingly change the picky E.CDECOMP rule:

$$(\langle x : c_1 \mapsto c_2 \rangle^{l_f}_{p_f} v_1) v_2 \rightarrow_{\text{indy}} \langle c_2 \{ x := (\langle c_1 \rangle^{l_f}_{p_f} v_2 \})^{l_f}_{p_f} (v_1 (\langle c_1 \rangle^{l_f}_{p_f} v_2)) \rangle^{l_f}_{p_f}$$

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 $\lambda_C$ is in the same position on the axis of blame as picky $\lambda_C$. We only need to change the labels on the contracts $\phi$ inserts to have an exact correspondence; however, $\psi$ will remain inexact.

Superpicky $\lambda_H$ reworks the F.CDECOMP rule in an attempt to harmonize $\lambda_H$ and picky $\lambda_C$ semantics:

$$((x : S_{11} \rightarrow S_{12} \Rightarrow x : S_{21} \rightarrow S_{22})^{l}_{w_2} w_2 \rightarrow_h \langle S_{12} \{ x := (S_{21} \Rightarrow S_{11})^{l}_{w_2} \} \Rightarrow \langle S_{22} \{ x := (S_{21} \Rightarrow S_{11})^{l}_{w_2} \} \rangle^{l}_{w_2}) \rangle^{l}_{w_2}$$

This seems to resolve the problem with $\psi$ into picky $\lambda_C$, but it poses problems in the proof of semantic-type soundness for $\lambda_H$: how do $S_{22}\{ x := w_2 \}$ and $S_{22}\{ x := (S_{11} \Rightarrow S_{21})^{l}_{w_2} \}$ 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—$\lambda_H$’s type semantics and the correspondences—to account for nontermination. We conjecture that step-indexing (Ahmed 2006) will suffice.

8 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).

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 CBV 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 and Flanagan (2010), which uses full $\beta$-reduction (more on this below).

---

6 This idea is due to Jeremy Siek (personal communication, January 2010).
Table 1. Comparison between contract systems

<table>
<thead>
<tr>
<th>Latent systems</th>
<th>FF02</th>
<th>HJL06</th>
<th>GF07</th>
<th>BM06</th>
<th>DFFF11</th>
<th>our $\lambda_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep (6)</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓ or either</td>
</tr>
<tr>
<td>eval order</td>
<td>CBV</td>
<td>lazy</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
</tr>
<tr>
<td>blame (8)</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
</tr>
<tr>
<td>checking (9)</td>
<td>if</td>
<td>if</td>
<td>active</td>
<td>active</td>
<td>active</td>
<td>active</td>
</tr>
<tr>
<td>typing (10)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>any con (11)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manifest systems</th>
<th>GF07 $\lambda_H$</th>
<th>F06</th>
<th>KF10</th>
<th>WF09</th>
<th>OTMW04</th>
<th>BGIP11</th>
<th>our $\lambda_H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep (6)</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>eval order</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
<td>CBV</td>
</tr>
<tr>
<td>blame (8)</td>
<td>$\uparrow l$</td>
<td>stuck</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
<td>$\uparrow l$</td>
</tr>
<tr>
<td>checking (9)</td>
<td>○</td>
<td>○</td>
<td>active</td>
<td>active</td>
<td>if</td>
<td>active</td>
<td>active</td>
</tr>
<tr>
<td>typing (10)</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>any con (11)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Notes: (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 text 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 (○)? (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 (×)?

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 Flanagan (2006) and Gronski and Flanagan (2007), the operational semantics checks casts “all in one go”:

$$s_2\{x := k\} \longrightarrow^*_h \text{true}$$

$$\langle\{x:B \mid s_1\} \Rightarrow \{x:B \mid s_2\}\rangle^l k \longrightarrow^*_h k$$
Such rules are formally awkward, and in any case they violate the spirit of a small-step semantics. Also, the formal definitions of $\lambda_H$ in both Flanagan (2006) and Gronski and Flanagan (2007) involve a circularity between the typing, subtyping, and implication relations. Knowles and Flanagan (2010) improve the technical presentation of $\lambda_H$ 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 $\phi$ and $\psi$ and their properties, and (2) the formulation and metatheory of dependent $\lambda_H$. (Dependent $\lambda_C$ is not a contribution on its own: many similar systems have been studied, and in any case its properties are simple.) The non-dependent part of our $\phi$ translation essentially coincides with the one studied by Gronski and Flanagan (2007), and our behavioral correspondence theorem is essentially the same as their theorem. Our $\psi$ translation completes their story for the non-dependent case, establishing a tight connection between the systems. The full dependent forms of $\phi$ and $\psi$ 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 $\lambda_H$ is most comparable to that of Knowles and 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 $\lambda_H$ of Knowles and Flanagan (2010) (though not the earlier one of Gronski and Flanagan (2007)), 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, nondeterministic $\beta$-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 CBV reduction because we consider blame as an exception—a computational effect—and we want 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 (2007), since CBV is a restriction of full $\beta$-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 $\{y:B \mid \text{true}\}$ is not a subtype of $\{y:B \mid (\lambda x:S. \text{true}) \text{ bad}\}$ because the contract evaluates to blame. However, the subtyping does hold in the Knowles and Flanagan system (2007) because the predicate reduces to true.

The system studied by Ou et al. (2004) is also close in spirit to our $\lambda_H$. 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 7—see the row labeled “any con”) allows arbitrary user-defined boolean functions to be used as contracts.
Our description of $\lambda_C$ is ultimately based on $\lambda_{\text{Con}}$ (Findler & Felleisen 2002), though our presentation is slightly different in its use of checks. Hinze et al. (2006) adapted Findler and Felleisen-style contracts (2002) to a location-passing implementation in Haskell, using picky-dependent function contracts.

Our $\lambda_H$-type semantics in Section 4.2 is effectively a semantics of contracts. Blume and McAllester (2006) offer 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 $\lambda_C$, 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” $\lambda_C$ and picky $\lambda_C$ 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 $\lambda_H$ 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 many papers on contracts and related ideas. We refer the reader to Knowles and Flanagan (2010) for a more comprehensive survey. Another useful resource is Wadler and Findler (2007) (technically superceded by Wadler and Findler (2007), 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 and Flanagan (2010) both support mixed static and dynamic checking of contracts, using, for example, a theorem prover. We have not addressed this aspect of their work, since we have chosen to work directly with the core calculus $\lambda_H$, which for them was the target of an elaboration function.

## 9 Conclusion

We can faithfully encode dependent $\lambda_H$ into $\lambda_C$—the behavioral correspondence is tight. $\lambda_H$’s F.CDECOMP rule forces us to accept a weaker behavioral correspondence when encoding $\lambda_C$ into $\lambda_H$, 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 non-dependent restriction.

## Acknowledgments

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


