Unifying Hybrid Types and Contracts

Jessica Gronski and Cormac Flanagan

University of California Santa Cruz

Abstract

Contract systems and hybrid type systems provide two alternative approaches for enforcing precisely-defined interface specifications, with complementary advantages: contract systems excel at blame assignment, whereas hybrid type systems support type-based static analysis.

We unify these two approaches by demonstrating that hybrid type checking is sufficiently expressive to encode higher-order contracts with proper blame assignment. In particular, a contract obligation that enforces both sides of a contract is decomposed into *two* type casts that each enforce one side of the contract. This expressiveness result provides several benefits, including allowing one of these casts to be lifted from variable references to variable definitions, resulting in improved contract coverage and removing the need for privileged contract obligations.

1 INTRODUCTION

The development of large software systems requires a modular development strategy where software modules communicate via well-understood interfaces. Ideally, these interfaces should be formally specified and mechanically enforced in order to detect, isolate, and localize software errors. Static type systems and dynamic contract systems [6] are two complementary approaches for enforcing software interfaces. Recent work on hybrid type checking [7] combines these two approaches, providing both the expressiveness benefits of contract systems while still verifying or refuting many properties at compile time, much like traditional type systems.

A key feature of modern contract systems is that they excel at *blame assignment*. Each contract obligation contains the labels of both modules that are party to that contract, so that the appropriate module can be blamed for any contract violation. Blame assignment works correctly even in the presence of complex control and data-flow operations involving higher-order functions, callbacks, etc. In large software systems, this ability to not only detect but also to localize software errors is extremely important.

For hybrid type systems, however, the analogous *type cast* operation contains only one module label (instead of two). This difference suggests that hybrid type systems are weaker at blame assignment, and, in this sense, fundamentally less expressive than contract systems [4].

This paper investigates the relationship between contract systems and hybrid type systems. We work in the context of two idealized languages: a contract language λ^C (based on λ^{Con} [6]) and the hybrid-typed language λ^H [7]. Surprisingly, we show that λ^H is sufficiently powerful to express all λ^C programs. In particular, the doubly-labelled contract obligation of λ^C (which enforces both sides of a contract) is equivalent to *two* singly-labelled type casts in λ^H (each of which enforces only one side of the contract). Prior work showed that contracts can naturally be implemented as pairs of projections, each with a single blame label [5]. This paper carries that development one step further, by showing that the type casts of λ^H exactly provide this projection functionality. Moreover, we believe that reifying these projections as a syntactic construct in the language provides additional flexibility and clarity.

In addition to our main expressiveness result, this connection between contract systems and hybrid type systems provides a formal foundation for understanding the relationship between these two approaches, which we hope will facilitate further cross-pollination between these domains. In particular, our result suggests that the static-analysis machinery of hybrid type systems (including recent results on the decidability of type inference [11]) could be applicable to λ^C contracts and programs, perhaps strengthening existing contract-based analyses [15].

An immediate benefit of expressing a λ^C contract as two λ^H casts is that it allows one of these casts to be lifted or *refactored* from the reference to an exported variable to the definition of that variable, which provides earlier error detection and improved contract coverage over λ^C.[1] This refactoring means that the type of each exported variable explicates the contract on that variable. In addition, this refactoring allows *privileged* contract obligations to be replaced by unprivileged type casts. That is, contract obligations are privileged in that a contract obligation in one module may need to assign blame to a different module, and so contract obligations should be inserted only by a trusted pre-processor or *elaborator*, and should not be present in source programs. A type cast is unprivileged if it only blames its containing module. In the refactored λ^H program, each module only contains unprivileged type casts that dynamically enforce that module's side of each contract. The λ^H type system then ensures, via assume-guarantee reasoning, that any assumption one module makes about a second module is guaranteed via dynamic type casts in that second module.

In the remainder of the paper, we first briefly review the two languages being compared, λ^C and λ^H, in Sections 2 and 3, respectively. Section 4 describes our translation from contracts to types. Section 5 shows how λ^H enables a notion of lifting that improves contract coverage. Section 6 proves the correctness of our translation. We conclude with a discussion of related work.

2 THE CONTRACT LANGUAGE λ^C

We begin by reviewing λ^C (see Figure 1), which extends the simply-typed lambda calculus with contracts along the lines of Findler and Felleisen [6]. The language is typed and includes both base types B (*Int* and *Bool*) and function types $T \to T$. In

[1]DrScheme's projection-based implementation of contracts includes a similar optimization [5].

Figure 1: λ^C Syntax and Evaluation Rules

$$\begin{array}{rcll}
B & ::= & Int \mid Bool & \textit{Base Types} \\
T & ::= & B \mid T \to T & \textit{Types} \\
c & ::= & \texttt{contract}\, B\, v \mid c \mapsto c & \textit{Contracts} \\
t & ::= & v \mid t\, t \mid \texttt{let}^l\, x : T : c = t\, \texttt{in}\, t \mid \texttt{blame}(l) \mid t^{c,l,l'} & \textit{Expressions} \\
v & ::= & x \mid k \mid \lambda x : T.t \mid v^{c\mapsto c,l,l'} & \textit{Values} \\
k & ::= & + \mid - \mid < \mid > \mid = \mid 0 \mid 1 \mid \dots & \textit{Constants}
\end{array}$$

Evaluation Contexts $\boxed{E}$

$$E = \bullet\, t \mid v\, \bullet \mid \texttt{let}^l\, x : T : c = \bullet\, \texttt{in}\, t \mid \bullet^{c,l,l'}$$

Evaluation Rules $\boxed{t \to_c t'}$

$$\begin{array}{rll}
k\, v \to_c [\![k]\!]\, (v) & & \text{[E-CONST]} \\
(\lambda x : T.t)\, v \to_c t[x := v] & & \text{[E-BETA]} \\
\texttt{let}^l\, x : T : c = v\, \texttt{in}\, t \to_c t[x := v] & & \text{[E-LET]} \\
v^{\texttt{contract}\, B\, v',l,l'} \to_c v & \text{if } v'\, v \to_c^* \texttt{true} & \text{[E-OK]} \\
v^{\texttt{contract}\, B\, v',l,l'} \to_c \texttt{blame}(l) & \text{if } v'\, v \to_c^* \texttt{false} & \text{[E-FAIL]} \\
v^{c\mapsto c',l,l'}\, v' \to_c [v\, (v'^{c,l',l})]^{c',l,l'} & & \text{[E-FUN]} \\
E[t_1] \to_c E[t_2] & \text{if } t_1 \to_c t_2 & \text{[E-COMPAT]} \\
E[\texttt{blame}(l)] \to_c \texttt{blame}(l) & & \text{[E-BLAME]}
\end{array}$$

addition to these simple types, the language includes contracts that more precisely define module interfaces[2]. A base contract ($\texttt{contract}\ B\ v$) describes the set of values of base type B that also satisfy the predicate v of type $B \to Bool$. A function contract $c \mapsto c'$ describes functions that take an argument satisfying contract c and return values satisfying contract c'.

A key goal of the contract system is to attribute blame for contract violations to particular program modules. To avoid complicating the language with a module system, λ^C modules are defined via let expressions. That is,

$$\texttt{let}^l\ x : T : c = t_1\ \texttt{in}\ t_2$$

defines a module t_1 identified by label l, that exports a variable x of type T and contract c. Code outside a let-bound expression is part of the *main* module.

Before execution, a λ^C program is first pre-processed (or *elaborated*) so that each reference to the above let-bound variable x from a different module l' is replaced by the *contract obligation* $x^{c,l,l'}$. A contract obligation enforces the contract c, and blames any contract violation on either the *server* module l that exports x or the *client* module l' that imports x.

The evaluation rules of λ^C programs are mostly straightforward: see Figure 1. The rule [E-CONST] relies on the auxiliary partial function $[\![k]\!] : \textit{Expression} \to \textit{Expression}$ to define the semantics of constant functions. For example, $[\![+]\!](3) =$

[2]Other languages, such as Eiffel [17], also include both types and contracts.

Figure 2: λ^C Type Rules

Type Environment $\boxed{\Gamma}$

$$\Gamma ::= \emptyset \mid \Gamma, x : T$$

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

$$[\text{T-VAR}]\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad [\text{T-CONST}]\ \frac{}{\Gamma \vdash k : ty(k)} \qquad [\text{T-LAM}]\ \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1 . t) : (T_1 \to T_2)}$$

$$[\text{T-APP}]\ \frac{\Gamma \vdash t_1 : (T_1 \to T_2) \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2} \qquad [\text{T-LET}]\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash_c c : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}^l\ x : T_1 : c = t_1\ \texttt{in}\ t_2 : T_2}$$

$$[\text{T-BLAME}]\ \frac{}{\Gamma \vdash \texttt{blame}(l) : T} \qquad [\text{T-OBLIG}]\ \frac{\Gamma \vdash t : T \quad \Gamma \vdash_c c : T}{\Gamma \vdash t^{c,l,l'} : T}$$

Contract Type Rules $\boxed{\Gamma \vdash_c c : T}$

$$[\text{T-BASEC}]\ \frac{\Gamma \vdash v : (B \to Bool)}{\Gamma \vdash_c \texttt{contract}\ B\ v : B} \qquad [\text{T-FUNC}]\ \frac{\Gamma \vdash_c c : T_1 \quad \Gamma \vdash_c c' : T_2}{\Gamma \vdash_c c \mapsto c' : (T_1 \to T_2)}$$

$+_3$ and $[\![+_3]\!](4) = 7$. The rules [E-BETA] and [E-LET] perform by-value evaluation of function applications and let expressions. The rule [E-COMPAT] compatibly closes the evaluation relation $\to_c$ over the evaluation context E.

The more interesting rules are those for evaluating a contract obligation $v^{c,l,l'}$. If c is a base contract ($\texttt{contract}\ B\ v'$) and ($v'\ v$) evaluates to `false`, then the contract obligation reduces, via [E-FAIL], to $\texttt{blame}(l)$, which blames the server l for providing an inappropriate value v for x. Otherwise, if ($v'\ v$) evaluates to `true`, then the contract is fulfilled and the rule [E-OK] removes the contract obligation from v. If c is a function contract $c \mapsto c'$, then $v^{c \mapsto c',l,l'}$ is considered a value. Once this value is applied to an argument, the obligation decomposes into two smaller obligations on the argument and the result of v, via [E-FUN].

The λ^C type system is defined via the usual judgement $\Gamma \vdash t : T$, which states that the expression t has type T in environment Γ. The auxiliary judgement $\Gamma \vdash_c c : T$ checks that the contract c is applicable to values of type T. The rules defining these two judgements are mostly straightforward: see Figure 2.

To illustrate the operational semantics of contracts, consider the following elaborated program P:

$\texttt{let}^f\ f : (Int \to Int) : (cNeg \mapsto cPos) = \lambda x : Int . x\ \texttt{in}$
$\texttt{let}^m\ m : Int : cPos =\ f^{cNeg \mapsto cPos,f,m}\ 4\ \texttt{in}\ m^{cPos,m,main}$

where

$$cPos = (\texttt{contract}\ Int\ (\lambda x : Int.\, x > 0))$$
$$cNeg = (\texttt{contract}\ Int\ (\lambda x : Int.\, x < 0))$$

This example includes two modules, f and m, where we label each module according to its exported variable. This program is evaluated as follows, where, for clarity, we shade each contract obligation grey or white, according to whether the obligation occurs on an expression produced by the module f or m, respectively.

$$\begin{aligned}
P &\rightarrow_c \texttt{let}^m\ m : Int : cPos = (\lambda x : Int.\, x)^{cNeg \mapsto cPos, f, m}\ 4\ \texttt{in}\ m^{cPos, m, main} \\
&\rightarrow_c \texttt{let}^m\ m : Int : cPos = [(\lambda x : Int.\, x)\ 4^{cNeg, m, f}]^{cPos, f, m}\ \texttt{in}\ m^{cPos, m, main} \\
&\rightarrow_c \texttt{let}^m\ m : Int : cPos = [(\lambda x : Int.\, x)\ \texttt{blame}(m)]^{cPos, f, m}\ \texttt{in}\ m^{cPos, m, main} \\
&\rightarrow_c \texttt{blame}(m)
\end{aligned}$$

In this case, the contract obligation $4^{cNeg,m,f}$ fails and the contract labels indicate that the error originated in module m, which violated the contract $cNeg \mapsto cPos$ of module f. Alternatively, if the literal 4 is replaced with -4, the first contract obligation would succeed and P would evaluate to:

$$\texttt{let}^m\ m : Int : cPos = -4^{cPos, f, m}\ \texttt{in}\ m^{cPos, m, main}$$

in this case blaming f for the violation. Although P includes only first-order functions, blame assignment also works in more complicated, higher-order situtations where functions are passed as arguments to other functions, etc.

3 THE HYBRID-TYPED λ^H CALCULUS

Whereas λ^C incorporates both a static type system and a dynamic contract system, λ^H [7] unifies these two interface specification systems into a single expressive type system.

The syntax of λ^H is shown in Figure 3 and includes types (S), expressions (s) and values (w). Types include refinement types of the form $\{x : B \mid s\}$, which describe the set of values of base type B that satisfy the predicate s. For example, $\{x : Int \mid x \geq 0\}$ describes the set of positive numbers. We sometimes use a base type B to abbreviate the trivial refinement type $\{x : B \mid \texttt{true}\}$. In λ^H, types can be enforced dynamically, via a type cast $\langle S_2 \Leftarrow S_1 \rangle^l\, s$. Here the expression s is statically typed as S_1 and the type cast dynamically enforces that the value produced by s also has type S_2; if not, the module labeled l is blamed.

Figure 3 defines the evaluation rules for λ^H; of particular interest are the rules for type casts. Casting a value w to a base refinement type $\{x : B \mid s\}$ involves checking if the predicate $s[x := w]$ evaluates to $\texttt{true}$.

A function cast $\langle (S_3 \rightarrow S_4) \Leftarrow (S_1 \rightarrow S_2) \rangle^l\ w$ is considered a value; once an argument w' is supplied, that cast is decomposed into two smaller casts on the function argument and result, via [F-FUN]:

$$(\langle (S_3 \rightarrow S_4) \Leftarrow (S_1 \rightarrow S_2) \rangle^l\ w)\ w' \rightarrow_h \langle S_4 \Leftarrow S_2 \rangle^l\, [w\ (\langle S_1 \Leftarrow S_3 \rangle^l\ w')]$$

Figure 3: λ^H Syntax

$$\begin{array}{rcll}
S & ::= & \{x{:}B \mid s\} \mid S \rightarrow S & \textit{Types} \\
s & ::= & w \mid s\,s \mid \texttt{let}^l\ x : S = s\ \texttt{in}\ s \mid \texttt{blame}(l) \mid \langle S \Leftarrow S \rangle^l\ s & \textit{Expressions} \\
w & ::= & x \mid k \mid \lambda x : S.\, s \mid \langle S \rightarrow S \Leftarrow S \rightarrow S \rangle^l\ w & \textit{Values}
\end{array}$$

Evaluation Contexts $\boxed{F}$

$$F ::= \bullet\ s \mid w\ \bullet \mid \texttt{let}^l\ x : S = \bullet\ \texttt{in}\ s \mid \langle S_2 \Leftarrow S_1 \rangle^l\ \bullet$$

Evaluation Rules $\boxed{s \longrightarrow_h s'}$

$$\begin{array}{rll}
k\ w \longrightarrow_h [\![k]\!]\ (w) & & [\text{F-Const}] \\
(\lambda x : S.\, s)\ w \longrightarrow_h s[x := w] & & [\text{F-Beta}] \\
\texttt{let}^l\ x : S = w\ \texttt{in}\ s \longrightarrow_h s[x := w] & & [\text{F-Let}] \\
\langle \{x{:}B \mid s_2\} \Leftarrow \{x{:}B \mid s_1\} \rangle^l\ w \longrightarrow_h w & \text{if } s_2[x := w] \longrightarrow_h^* \texttt{true} & [\text{F-OK}] \\
\langle \{x{:}B \mid s_2\} \Leftarrow \{x{:}B \mid s_1\} \rangle^l\ w \longrightarrow_h \texttt{blame}(l) & \text{if } s_2[x := w] \longrightarrow_h^* \texttt{false} & [\text{F-Fail}] \\
(\langle (S_3 \rightarrow S_4) \Leftarrow (S_1 \rightarrow S_2) \rangle^l\ w)\ w' \longrightarrow_h \langle S_4 \Leftarrow S_2 \rangle^l\ [w\ (\langle S_1 \Leftarrow S_3 \rangle^l\ w')] & & [\text{F-Fun}] \\
\langle (S_1 \rightarrow S_2) \Leftarrow \{x{:}B \mid s_1\} \rangle^l\ w \longrightarrow_h \texttt{blame}(l) & & [\text{F-Bad1}] \\
\langle \{x{:}B \mid s_1\} \Leftarrow (S_1 \rightarrow S_2) \rangle^l\ w \longrightarrow_h \texttt{blame}(l) & & [\text{F-Bad2}] \\
F[s_1] \longrightarrow_h F[s_2] & \text{if } s_1 \longrightarrow_h s_2 & [\text{F-Ctx}] \\
F[\texttt{blame}(l)] \longrightarrow_h \texttt{blame}(l) & & [\text{F-Blame}]
\end{array}$$

Function casts involve a subtle mix of static and dynamic reasoning. Since the original function w has type $S_1 \rightarrow S_2$, the evaluation rules must ensure that w is only applied to values of type S_1. The argument cast generated by [F-FUN], $\langle S_1 \Leftarrow S_3 \rangle^l\ w'$, relies on the type system to ensure that w' has type S_3, and then dynamically casts w' to a value of type S_1. Thus, in the function cast $\langle (S_3 \rightarrow S_4) \Leftarrow (S_1 \rightarrow S_2) \rangle^l\ w$, the types S_1 and S_4 are enforced dynamically, whereas the type system is responsible for enforcing S_2 and S_3.

The λ^H type system is defined via the judgement $\Delta \vdash s : S$, which states that the expression s has type S in environment Δ: see Figure 4. The auxiliary judgement $\Delta \vdash S$ checks that S is a well-formed type. Subtyping between function types is straightforward, via [SUB-FUN]. Subtyping between refinement types is defined via [SUB-BASE]. This rule uses the auxiliary judgement $\Delta \vdash s_1 \Rightarrow s_2$, which states that s_2 is true whenever s_1 is, in any variable substitution that is consistent with the type environment Δ. This notion of consistency between a substitution σ (from variables to values) with an environment Δ is formalized via the final judgement $\Delta \models \sigma$, and we refer the interested reader to [7] for more details.

Type checking for λ^H is in general undecidable. In this paper, however, since we start with well-typed λ^C programs and translate expressions in a manner that is type-preserving, our generated λ^H programs are well-typed by construction.

Figure 4: λ^H Type Rules

Type Environment $\boxed{\Delta}$

$$\Delta ::= \emptyset \mid \Delta, x : S$$

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

$$[\textsc{S-Var}]\ \frac{x : S \in \Delta}{\Delta \vdash x : S} \qquad [\textsc{S-Const}]\ \frac{}{\Delta \vdash k : ty(k)} \qquad [\textsc{S-Lam}^a]\ \frac{\Delta \vdash S_1 \quad \Delta, x : S_1 \vdash s : S_2 \quad x \notin FV(S_2)}{\Delta \vdash (\lambda x : S_1 . s) : (S_1 \to S_2)}$$

$$[\textsc{S-App}]\ \frac{\Delta \vdash s_1 : (S_1 \to S_2) \quad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\ s_2 : S_2} \qquad [\textsc{S-Let}]\ \frac{\Delta \vdash s_1 : S_1 \quad \Delta, x : S_1 \vdash s_2 : S_2[x := s_1]}{\Delta \vdash \mathtt{let}^l\ x : S_1 = s_1\ \mathtt{in}\ s_2 : S_2[x := s_1]}$$

$$[\textsc{S-Blame}]\ \frac{\Delta \vdash S}{\Delta \vdash \mathtt{blame}(l) : S} \qquad [\textsc{S-Cast}]\ \frac{\Delta \vdash S_2 \quad \Delta \vdash s : S_1}{\Delta \vdash \langle S_2 \Leftarrow S_1 \rangle^l\ s : S_2} \qquad [\textsc{S-Sub}]\ \frac{\Delta \vdash s : S_1 \quad \Delta \vdash S_2 \quad \Delta \vdash S_1 <: S_2}{\Delta \vdash s : S_2}$$

Well-Formed Types $\boxed{\Delta \vdash S}$

$$[\textsc{WF-Fun}]\ \frac{\Delta \vdash S_1 \qquad \Delta \vdash S_2}{\Delta \vdash (S_1 \to S_2)} \qquad [\textsc{WF-Base}]\ \frac{\Delta, x : B \vdash s : Bool}{\Delta \vdash \{x{:}B \mid s\}}$$

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

$$[\textsc{Sub-Fun}]\ \frac{\Delta \vdash S_3 <: S_1 \quad \Delta \vdash S_2 <: S_4}{\Delta \vdash (S_1 \to S_2) <: (S_3 \to S_4)} \qquad [\textsc{Sub-Base}]\ \frac{\Delta, x : B \vdash s_1 \Rightarrow s_2}{\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}}$$

Implication Rule $\boxed{\Delta \vdash s_1 \Rightarrow s_2}$

$$[\textsc{Imp}]\ \frac{\forall \sigma.\ (\Delta \models \sigma \text{ and } \sigma(s_1) \to_h^* \mathtt{true} \text{ implies } \sigma(s_2) \to_h^* \mathtt{true})}{\Delta \vdash s_1 \Rightarrow s_2}$$

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

$$[\textsc{CS-Empty}]\ \frac{}{\emptyset \models \emptyset} \qquad [\textsc{CS-Ext}]\ \frac{\emptyset \vdash s : S \quad (x := s)\Delta \models \sigma}{x : S, \Delta \models (x := s)\sigma}$$

[a]The hygiene condition $x \notin FV(S_2)$ was omitted from the initial version of this paper.

4 EXPRESSING CONTRACTS AS TYPES

We now show that λ^H is sufficiently powerful to express all λ^C programs, including proper attribution of contract violations to appropriate modules.

We begin by translating λ^C types (T) into more expressive λ^H types (S) via the translation $\phi_t : T \to S$, which adds the trivial refinement predicate `true` to base types.

$$\begin{aligned}\phi_t(B) &= \{x{:}B \mid \texttt{true}\}\\ \phi_t(T_1 \to T_2) &= \phi_t(T_1) \to \phi_t(T_2)\end{aligned}$$

We also translate λ^C contracts (c) into λ^H types (S) via the translation $\phi_c : c \to S$, which uses refinement types to emulate contracts.

$$\begin{aligned}\phi_c(\texttt{contract}\, B\, v) &= \{x{:}B \mid \phi(v)\, x\} \qquad \text{if } x \notin FV(v)\\ \phi_c(c \mapsto c') &= \phi_c(c) \to \phi_c(c')\end{aligned}$$

These two translations already shed some light on the relationship between λ^C types and contracts. Suppose that c is a contract over some type T (i.e. $\Gamma \vdash_c c : T$). Since the contract c is a "restriction" of T, we might expect that $\phi_c(c)$ would be a subtype of $\phi_t(T)$. This property holds for base contracts, but not for function contracts, because of the contravariance of function domains. Thus $\phi_c(c)$ and $\phi_t(T)$ may be incomparable under the subtyping relation. The two λ^H types $\phi_c(c)$ and $\phi_t(T)$ are, however, identical in structure, except that $\phi_c(c)$ has more precise refinement predicates. That is, $base(\phi_c(c)) = \phi_t(T)$, where $base : S \to S$ is a function that strips refinement predicates from λ^H types.

$$\begin{aligned}base(\{x{:}B \mid s\}) &= \{x{:}B \mid \texttt{true}\}\\ base(S_1 \to S_2) &= base(S_1) \to base(S_2)\end{aligned}$$

Finally, we consider how to translate λ^C expressions into behaviorally-equivalent λ^H expressions, and in particular, how to translate the contract obligation in our earlier example P:

$$f^{cNeg \mapsto cPos, f, m}\ 4$$

Recall that f has type $Int \to Int$. The translated expression $\phi(f)$ should then have type $sInt \to sInt$, where $sInt$ is the trivial refinement type $\{x{:}Int \mid \texttt{true}\}$.

As a first attempt we could translate the above contract obligation into a corresponding type cast, yielding the λ^H expression

$$[\langle S \Leftarrow base(S)\rangle^f\, f]\ 4$$

where

$$\begin{aligned}S &= (sNeg \to sPos) & sNeg &= \phi_c(cNeg)\\ base(S) &= (sInt \to sInt) & sPos &= \phi_c(cPos)\end{aligned}$$

This translation has two problems, however. First, it retains only one blame label, resulting in incorrect blame assignment. Second, and more importantly, the translated program is ill-typed, since the casted function has type $S = sNeg \to sPos$, and so should not be applied to 4, an expression of type $sInt$. Thus, this program is both ill-typed and also no longer enforces the original $cNeg$ domain contract.

To solve both these problems, we introduce a second type cast to dynamically enforce the domain part of function contracts, yielding the translated λ^H program:

$$[\langle base(S) \Leftarrow S\rangle^m\,(\langle S \Leftarrow base(S)\rangle^f\, f)]\ 4$$

Thus, given f of type $base(S) = (sInt \to sInt)$, the first type cast $\langle S \Leftarrow base(S)\rangle^f$ dynamically ensures the covariant property, that the function only returns positive integers, but relies on the type system to ensure the contravariant property, that the casted function is only applied to negative integers. The second type cast $\langle base(S) \Leftarrow S\rangle^m$ dynamically fulfills this second obligation, producing a function of type $(sInt \to sInt)$ that statically may be applied to any integer, but dynamically fails and blames module m if ever given non-negative arguments. Together, these two type casts exactly enforce the semantics of the original λ^C contract obligation, with correct blame assignment.

The complete translation $\phi : t \to s$ from λ^C expressions to λ^H expressions is then the compatible closure of this rule for translating contract obligations:

$$\begin{aligned}
\phi(t^{c,l,l'}) &= \langle base(S) \Leftarrow S\rangle^{l'}\,\langle S \Leftarrow base(S)\rangle^{l}\,\phi(t) \quad \text{where } S = \phi_c(c)\\
\phi(x) &= x\\
\phi(k) &= k\\
\phi(\lambda x : T.\,t) &= \lambda x : \phi_t(T).\,\phi(t)\\
\phi(t_1\ t_2) &= \phi(t_1)\ \phi(t_2)\\
\phi(\texttt{let}^l\ x : T : c = t_1\ \texttt{in}\ t_2) &= \texttt{let}^l\ x : \phi_t(T) = \phi(t_1)\ \texttt{in}\ \phi(t_2)\\
\phi(\texttt{blame}(l)) &= \texttt{blame}(l)
\end{aligned}$$

4.1 Example

To illustrate this translation, consider the translation of the earlier example $\phi(P)$:

$$\begin{array}{l}
\texttt{let}^f\ f : (sInt \to sInt) = \lambda x : sInt.\,x\ \texttt{in}\\
\texttt{let}^m\ m : sInt = [\langle base(S) \Leftarrow S\rangle^m\,\langle S \Leftarrow base(S)\rangle^f\, f]\ 4\ \texttt{in}\\
\langle sInt \Leftarrow sPos\rangle^{main}\,\langle sPos \Leftarrow sInt\rangle^m\ m
\end{array}$$

where the λ^H type $S = sNeg \to sPos$ encodes the original contract. Note that $\phi(P)$ mostly includes only simple types; precise refinement types are only used in casts to implement contract obligations in the original program. These precise refinement types occur in matched pairs such as $\langle sInt \Leftarrow sPos\rangle^{main}\,\langle sPos \Leftarrow sInt\rangle^m\ m$, where the precise type $sPos$ is first dynamically enforced and then statically assumed. Note that the base contract $cPos$ on module m has no contravariant component, and so the second resulting type cast $\langle sInt \Leftarrow sPos\rangle^{main}$ is actually an up-cast, and could be optimized away. That is, the second cast is only necessary for function contracts that involve bidirectional communication between modules.

The program $\phi(P)$ evaluates as follows, correctly blaming m:

$$\begin{array}{l}
\longrightarrow_h \texttt{let}^m\ m : \mathit{sInt} =\ [\langle \mathit{base}(S) \Leftarrow S\rangle^m\ \langle S \Leftarrow \mathit{base}(S)\rangle^f (\lambda x : \mathit{sInt}.x)]\ 4\ \texttt{in}\ \ldots \\
\longrightarrow_h \texttt{let}^m\ m : \mathit{sInt} = \\
\qquad \langle \mathit{sInt} \Leftarrow \mathit{sPos}\rangle^m\ ([\langle S \Leftarrow \mathit{base}(S)\rangle^f (\lambda x : \mathit{sInt}.x)]\ \langle \mathit{sNeg} \Leftarrow \mathit{sInt}\rangle^m 4)\ \texttt{in}\ \ldots \\
\longrightarrow_h \texttt{let}^m\ m : \mathit{sInt} =\ \langle \mathit{sInt} \Leftarrow \mathit{sPos}\rangle^m\ ([\langle S \Leftarrow \mathit{base}(S)\rangle^f (\lambda x : \mathit{sInt}.x)]\ \texttt{blame}(m))\ \ldots \\
\longrightarrow_h \texttt{blame}(m)
\end{array}$$

Conversely, if the literal 4 were replaced with -4, then $\phi(P)$ would evaluate as follows, blaming f:

$$\texttt{let}^m\ m : \mathit{sInt} =\ \langle \mathit{sInt} \Leftarrow \mathit{sPos}\rangle^m\ \langle \mathit{sPos} \Leftarrow \mathit{sInt}\rangle^f\ (-4)\ \texttt{in}\ \ldots$$

5 IMPROVING CONTRACT COVERAGE

In the above program $\phi(P)$, every reference to an exported variable, such as f, is enclosed in a cast, such as $\langle S \Leftarrow \mathit{base}(S)\rangle^f\ f$, which does not mention the client module. Hence, we could refactor this program to avoid repeatedly re-checking these casts at each reference of an exported variable, and instead check those casts only once, as variables are exported.

$$\begin{array}{l}
\texttt{let}^f\ f : S =\ \langle S \Leftarrow \mathit{base}(S)\rangle^f\ (\lambda x : \mathit{sInt}.x)\ \texttt{in} \\
\texttt{let}^m\ m : \mathit{sPos} =\ \langle \mathit{sPos} \Leftarrow \mathit{sInt}\rangle^m\ [(\langle \mathit{base}(S) \Leftarrow S\rangle^m\ f)\ 4]\ \texttt{in}\ m
\end{array}$$

In this refactored program (with up-casts optimized away), the "contract" on the exported variable f is explicated via the precise type $S = \mathit{sNeg} \rightarrow \mathit{sPos}$; this precise type is enforced via the dynamic type cast $\langle S \Leftarrow \mathit{base}(S)\rangle^f$ inside module f. The second cast $\langle \mathit{base}(S) \Leftarrow S\rangle^m\ f$ remains in the client module m to detect attempts by m to pass incorrect arguments to f.

This refactoring yields two main advantages. First, evaluating the cast $\langle S \Leftarrow \mathit{base}(S)\rangle^f$ only once instead of multiple times may result in better performance. Second, the refactoring provides earlier, and hence better, error detection.

To illustrate this idea, consider the following λ^C program, where the module f exports the literal 4, in clear violation of its *cNeg* contract:

$$P' = \texttt{let}^f\ f : \mathit{Int} : \mathit{cNeg} =\ 4\ \texttt{in}\ \ldots\ (f^{\mathit{cNeg},f,\mathit{main}})\ \ldots$$

In both this program and the corresponding λ^H program $\phi(P')$, the contract violation is not actually detected unless the contract obligation $f^{\mathit{cNeg},f,\mathit{main}}$ is evaluated, which may of course only happen when certain code paths are exercised. In contrast, the refactored λ^H version of this program:

$$\texttt{let}^f\ f : \mathit{sNeg} =\ \langle \mathit{sNeg} \Leftarrow \mathit{sInt}\rangle^f\ 4\ \texttt{in}\ \ldots\ \langle \mathit{sInt} \Leftarrow \mathit{sNeg}\rangle^{\mathit{main}}\ f\ \ldots$$

would detect this error immediately. Thus, the refactoring enabled by the translation to λ^H permits somewhat increased *contract coverage* over λ^C.

6 CORRECTNESS OF THE TRANSLATION

We now prove that ϕ is a semantics preserving translation. First, ϕ is type-preserving in that it maps well-typed λ^C expressions to well-typed λ^H expressions. The formal statement and proof of this property relies on an auxiliary function ϕ_Γ that maps λ^C type environments to λ^H type environments by translating the types in the bindings.

$$\begin{aligned}\phi_\Gamma(\emptyset) &= \emptyset \\ \phi_\Gamma(\Gamma, x : T) &= \phi_\Gamma(\Gamma), x : \phi_t(T)\end{aligned}$$

Theorem 1 (Type Preservation)

1. *If* $\Gamma \vdash t : T$ *then* $\phi_\Gamma(\Gamma) \vdash \phi(t) : \phi_t(T)$
2. *If* $\Gamma \vdash_c c : T$ *then* $\phi_\Gamma(\Gamma) \vdash \phi_c(c)$

Second, the translation ϕ also preserves the operational semantics in that if a λ^C expression t evaluates to t', then the translation $\phi(t)$ evaluates in λ^H to some expression s that is also reachable from $\phi(t')$.

Theorem 2 (Behavioral Equivalence) *If* $t \longrightarrow_c^* t'$ *then* $\exists s$ *such that* $\phi(t) \longrightarrow_h^* s$ *and* $\phi(t') \longrightarrow_h^* s$.

7 RELATED AND FUTURE WORK

The enforcement of complex program specifications, or *contracts*, is the subject of a large body of prior work [16, 6, 13, 9, 10, 14, 18, 12, 5, 2]. Since these contracts are typically not expressible in classical type systems, they have previously been relegated to dynamic checking, as in, for example, Eiffel [16]. Eiffel's expressive contract language is strictly separated from its type system. The Bigloo Scheme compiler [19] introduced higher-order contracts. Findler and Felleisen [6] describe λ^{Con}, a language with a higher-order contract system that provides an elegant way to introduce, propagate, and enforce contracts, and to assign blame appropriately. Blume and McAllester [1] model λ^{Con} in order to prove the soundness of the contract checker and extend the system with recursive contracts.

Work on advanced type systems influenced our choice of how to express program invariants in λ^H. In particular, Freeman and Pfenning [8] extended ML with another form of refinement types. Their work focuses on providing both decidable type checking and type inference, instead of on supporting arbitrary refinement predicates. Xi and Pfenning have explored applications of dependent types in Dependent ML [21, 20]. In a complementary approach, Chen and Xi [3] address decidability limitations by providing a mechanism through which the programmer can provide proofs of subtle properties in the source code.

This paper attempts to connect these two fields of study by developing a formal connection between type systems and contract systems, and by showing how hybrid type system can express higher-order contracts obligations with precise blame assignment.

Recently, Meunier et al investigated statically verifying contracts via set-based analysis [15], and Findler and Blume [5] defined a partial-order for higher-order contracts that is contravariant in the domain. These ideas appear closely related to the corresponding notions of type inference and subtyping in type theory. We plan to use the framework of this paper to formally explore these kinds of deep connections between contract theory and type theory. Further cross-pollination between these two fields may yield additional contributions to type theory, and may also help apply existing type theory to contract-based programs.

Acknowledgements We would like to thank Robby Findler and Matthias Felleisen for valuable feedback on this paper. This work was supported by a Sloan Fellowship and by NSF grant CCR-0341179.

REFERENCES

[1] M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

[2] M. Blume and D. A. McAllester. A sound (and complete) model of contracts. In *ICFP*, pages 189–200, 2004.

[3] C. Chen and H. Xi. Combining programming with theorem proving. In *ICFP*, pages 66–77, 2005.

[4] R. Findler. Personal communication, October 2006.

[5] R. B. Findler and M. Blume. Contracts as pairs of projections. In M. Hagiya and P. Wadler, editors, *FLOPS '06*, volume 3945. Springer, 2006.

[6] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, New York, NY, USA, 2002. ACM Press.

[7] C. Flanagan. Hybrid type checking. In *POPL '06*, pages 245–256, New York, NY, USA, 2006. ACM Press.

[8] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[9] B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. A language manual for Sather 1.1, 1996.

[10] R. C. Holt and J. R. Cordy. The Turing programming language. *Commun. ACM*, 31:1310–1424, 1988.

[11] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP '07 (to appear)*, 2007.

[12] M. Kölling and J. Rosenberg. Blue: Language specification, version 0.94, 1997.

[13] G. T. Leavens and Y. Cheon. Design by contract with JML, 2005. avaiable at `http://www.cs.iastate.edu/~leavens/JML/`.

[14] D. Luckham. Programming with specifications. *Texts and Monographs in Computer Science*, 1990.

[15] P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 218–231, 2006.

[16] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[17] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[18] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.

[19] M. Serrano. *Bigloo: A practical Scheme Compiler*, 1992-2002.

[20] H. Xi. Imperative programming with dependent types. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 214–227, 1999.

A PROOF OF TYPE PRESERVATION

The proof of type preservation relies on following two simple lemmas:

Lemma 3 (Well-formed Translations) $\forall \Delta, T.\ \Delta \vdash \phi_t(T)$

Lemma 4 *If* $\Gamma \vdash_c c : T$ *then* $base(\phi_c(c)) = \phi_t(T)$

Assumption 5 *Let* ty^C *and* ty^H *be the constant typing functions for* λ^C *and* λ^H *respectively. We assume that* $\forall k.\ ty^H(k) <: \phi_t(ty^C(k))$*. Also, following [7], we assume that each basic constant is assigned a singleton type that denotes exactly that constant. For example,* $ty^H(3) = \{x{:}Int \,|\, x = 3\}$*.*

Restatement of Theorem 1 (Type Preservation)

1. If $\Gamma \vdash t : T$ *then* $\phi_\Gamma(\Gamma) \vdash \phi(t) : \phi_t(T)$

2. If $\Gamma \vdash_c c : T$ *then* $\phi_\Gamma(\Gamma) \vdash \phi_c(c)$

PROOF: The proof is an induction on the type derivation for $\Gamma \vdash t : T$ and $\Gamma \vdash_c c : T$.

1. [T-LAM] Suppose $\Gamma \vdash (\lambda x : T_1 . t) : (T_1 \to T_2)$. Then $\Gamma, x : T_1 \vdash t : T_2$.
   By induction, $\phi_\Gamma(\Gamma), x : \phi_t(T_1) \vdash \phi(t) : \phi_t(T_2)$.
   By Lemma 3, $\phi_\Gamma(\Gamma) \vdash \phi_t(T_1)$.
   Hence, $\phi_\Gamma(\Gamma) \vdash \phi(\lambda x : T_1 . t) : (\phi_t(T_1) \to \phi_t(T_2))$ via [S-LAM].

   [T-CONST] Suppose $\Gamma \vdash k : ty^C(k)$.
   By Assumption 5, $\phi_\Gamma(\Gamma) \vdash k : ty^H(k)$, via [S-CONST].

   [T-VAR] Suppose $\Gamma \vdash x : T$. Then $x : T \in \Gamma$.
   By construction, $x : \phi_t(T) \in \phi_\Gamma(\Gamma)$.
   Hence, $\phi_\Gamma(\Gamma) \vdash x : \phi_t(T)$ via [S-VAR].

[T-APP] Suppose $\Gamma \vdash t_1\ t_2 : T_2$. Then $\Gamma \vdash t_1 : (T_1 \rightarrow T_2)$ and $\Gamma \vdash t_2 : T_1$.
By induction, $\phi_\Gamma(\Gamma) \vdash \phi(t_1) : (\phi_t(T_1) \rightarrow \phi_t(T_2))$ and $\phi_\Gamma(\Gamma) \vdash \phi(t_2) : \phi_t(T_1)$.
Hence, $\phi_\Gamma(\Gamma) \vdash \phi(t_1\ t_2) : \phi_t(T_2)$ via [S-APP].

[T-LET] Suppose $\Gamma \vdash \texttt{let}^l\ x : T_1 : c = t_1\ \texttt{in}\ t_2 : T_2$. Then $\Gamma \vdash t_1 : T_1$ and $\Gamma, x : T_1 \vdash t_2 : T_2$.
By induction, $\phi_\Gamma(\Gamma) \vdash \phi(t_1) : \phi_t(T_1)$ and $\phi_\Gamma(\Gamma), x : \phi_t(T_1) \vdash \phi(t_2) : \phi_t(T_2)$.
Hence, $\phi_\Gamma(\Gamma) \vdash \phi(\texttt{let}^l\ x : T_1 : c = t_1\ \texttt{in}\ t_2) : \phi_t(T_2)$, via [S-LET] since $x \notin FV(\phi_t(T_2))$.

[T-BLAME] Suppose $\Gamma \vdash \texttt{blame}(l) : T$. Then $\phi_\Gamma(\Gamma) \vdash \texttt{blame}(l) : \phi_t(T)$ via [S-BLAME] and Lemma 3.

[T-OBLIG] Suppose $\Gamma \vdash t^{c,l_1,l_2} : T$. Then $\Gamma \vdash t : T$ and $\Gamma \vdash_c c : T$.
Let $S = \phi_c(c)$. By Lemma 4, $base(S) = \phi_t(T)$.
Hence, $\phi_\Gamma(\Gamma) \vdash \phi(t) : base(S)$ and $\phi_\Gamma(\Gamma) \vdash S$ by induction.
By Lemma 3, $\phi_\Gamma(\Gamma) \vdash base(S)$.
Hence via two applications of [S-CAST],
$\phi_\Gamma(\Gamma) \vdash [\langle base(S) \Leftarrow S\rangle^{l_2}\ \langle S \Leftarrow base(S)\rangle^{l_1}\ \phi(t)] : base(S)$
Or equivalently $\phi_\Gamma(\Gamma) \vdash \phi(t^{c,l_1,l_2}) : \phi_t(T)$.

2. [T-BASEC] Suppose $\Gamma \vdash_c \texttt{contract}\ B\ v : B$. Then $\Gamma \vdash v : (B \rightarrow Bool)$.
By induction $\phi_\Gamma(\Gamma) \vdash \phi(v) : (B \rightarrow Bool)$.
Let $x \notin FV(\phi(v))$. Then $\phi_\Gamma(\Gamma), x : B \vdash \phi(v) : (B \rightarrow Bool)$.
By [S-FUN], $\phi_\Gamma(\Gamma), x : B \vdash \phi(v)\ x : Bool$.
By [WF-BASE], $\phi_\Gamma(\Gamma) \vdash \{x{:}B \mid \phi(v)\ x\}$ or $\phi_\Gamma(\Gamma) \vdash \phi_c(\texttt{contract}\ B\ v)$.

[T-FUNC] Suppose $\Gamma \vdash_c c \mapsto c' : (T_1 \rightarrow T_2)$. Then $\Gamma \vdash_c c : T_1$ and $\Gamma \vdash_c c' : T_2$.
By induction $\phi_\Gamma(\Gamma) \vdash \phi_c(c)$and $\phi_\Gamma(\Gamma) \vdash \phi_c(c')$.
By [WF-FUN] $\phi_\Gamma(\Gamma) \vdash \phi_c(c) \rightarrow \phi_c(c')$ or $\phi_\Gamma(\Gamma) \vdash \phi_c(c \mapsto c')$.

□

B PROOF OF BEHAVIORAL EQUIVALENCE

The proof of behavioral equivalence relies on the following lemmas:

Lemma 6 $\phi(t_1[x := t_2]) = \phi(t_1)[x := \phi(t_2)]$

Lemma 7 *The translation ϕ maps λ^C values to λ^H values.*

Assumption 8 *Let $[\![\cdot]\!]^C$ and $[\![\cdot]\!]^H$ define the semantics of primitive functions k for λ^C and λ^H respectively. We assume each primitive function k behaves equivalently on any λ^C value v and on the corresponding λ^H value $\phi(v)$,* i.e., $\phi([\![k]\!]^C\ v) = [\![k]\!]^H\ \phi(v)$.

Lemma 9 (Single Step Behavioral Equivalence) *If $t \rightarrow_c t'$ then $\exists s.\, \phi(t) \rightarrow_h^* s$ and $\phi(t') \rightarrow_h^* s$.*

PROOF: The proof is by induction on the derivation of $t \rightarrow_c t'$. For all cases except the [E-FUN] case, $s = \phi(t')$.

[E-CONST] Suppose $k\ v \rightarrow_c [\![k]\!]\ (v)$.

$$\begin{aligned} \phi(k\ v) &= k\ \phi(v) \\ &\rightarrow_h [\![k]\!]^H\ \phi(v) \text{ by Lemma 7 and [F-CONST].} \\ &= \phi([\![k]\!]^C\ v) \text{ by Assumption 8.} \end{aligned}$$

[E-BETA] Suppose $(\lambda x : T.\, t)\ v \rightarrow_c t[x := v]$.

$$\begin{aligned} \phi((\lambda x : T.\, t)\ v) &= (\lambda x : \phi_t(T).\, \phi(t))\ \phi(v) \\ &\rightarrow_h \phi(t)[x := \phi(v)] \text{ by Lemma 7 and [F-BETA].} \\ &= \phi(t[x := v]) \text{ by Lemma 6.} \end{aligned}$$

[E-LET] Suppose $\texttt{let}^l\ x : T : c =\ v\ \texttt{in}\ t' \rightarrow_c t'[x := v]$.

$$\begin{aligned} \phi(\texttt{let}^l\ x : T : c =\ v\ \texttt{in}\ t') &= \texttt{let}^l\ x : \phi_t(T) =\ \phi(v)\ \texttt{in}\ \phi(t') \\ &\rightarrow_h \phi(t')[x := \phi(v)] \text{ by Lemma 7 and [F-LET].} \\ &= \phi(t'[x := v]) \text{ by Lemma 6.} \end{aligned}$$

[E-BLAME] Suppose $E[\texttt{blame}(l)] \rightarrow_c \texttt{blame}(l)$.

$$\begin{aligned} \text{For some } F,\ \phi(E[\texttt{blame}(l)]) &= F[\texttt{blame}(l)] \\ &\rightarrow_h \texttt{blame}(l) \text{ by [F-BLAME].} \\ &= \phi(\texttt{blame}(l)) \text{ by definition.} \end{aligned}$$

[E-OK] Suppose $v^{\texttt{contract}\ B\ v', l_1, l_2} \rightarrow_c v$ and $v'\ v \rightarrow_c^* \texttt{true}$.

$$\begin{aligned} &\phi([v^{\texttt{contract}\ B\ v', l_1, l_2}]) \\ &= \langle \phi_t(B) \Leftarrow \{x{:}\phi_t(B) \mid \phi(v')\,x\}\rangle^{l_2} \langle \{x{:}\phi_t(B) \mid \phi(v')\,x\} \Leftarrow \phi_t(B)\rangle^{l_1} \phi(v) \\ &\rightarrow_h \langle \phi_t(B) \Leftarrow \{x{:}\phi_t(B) \mid \phi(v')\,x\}\rangle^{l_2}\ \phi(v) \text{ by [F-OK]} \\ &\qquad\qquad \text{because of Lemma 7 and } \phi(v'\ v) \rightarrow_c^* \texttt{true} \text{ by induction.} \\ &\rightarrow_h \phi(v) \end{aligned}$$

[E-FAIL] Suppose $v^{\texttt{contract}\ B\ v', l_1, l_2} \rightarrow_c \texttt{blame}(l)$ and $v'\ v \rightarrow_c^* \texttt{false}$.

$$\begin{aligned} &\phi([v^{\texttt{contract}\ B\ v', l_1, l_2}]) \\ &= \langle \phi_t(B) \Leftarrow \{x{:}\phi_t(B) \mid \phi(v')\,x\}\rangle^{l_2} \langle \{x{:}\phi_t(B) \mid \phi(v')\,x\} \Leftarrow \phi_t(B)\rangle^{l_1} \phi(v) \\ &\rightarrow_h \langle \phi_t(B) \Leftarrow \{x{:}\phi_t(B) \mid \phi(v')\,x\}\rangle^{l_2}\ \texttt{blame}(l_1) \text{ by [F-FAIL]} \\ &\qquad\qquad \text{by Lemma 7 and since } \phi(v'\ v) \rightarrow_c^* \texttt{false} \text{ by induction.} \\ &\rightarrow_h \texttt{blame}(l_1) \text{ by [F-BLAME]} \\ &= \phi(\texttt{blame}(l_1)). \end{aligned}$$

[E-FUN] Suppose $[v^{c \mapsto c', p, n}\ v'] \rightarrow_c [(v\ v'^{c,n,p})^{c',p,n}]$. Let $S = \phi_c(c)$ and $S' = \phi_c(c')$. and $s_f = (\langle S \Leftarrow \mathit{base}(S)\rangle^n\ \phi(v'))$.

$$\begin{aligned} & \phi([v^{c \mapsto c', p, n}\ v']) \\ = \quad & [\langle \mathit{base}(S \rightarrow S') \Leftarrow (S \rightarrow S')\rangle^n \langle (S \rightarrow S') \Leftarrow \mathit{base}(S \rightarrow S')\rangle^p\, \phi(v)]\ \phi(v') \\ \rightarrow_h \quad & \langle \mathit{base}(S') \Leftarrow S'\rangle^n\, [[\langle (S \rightarrow S') \Leftarrow \mathit{base}(S \rightarrow S')\rangle^p\, \phi(v)]\ s_f] \end{aligned}$$

If S is a function type, then s_f is a value, and so:

$$\begin{array}{ll} & \langle base(S') \Leftarrow S'\rangle^n\,[[\langle(S \to S') \Leftarrow base(S \to S')\rangle^p\,\phi(v)]\; s_f] \\ \to_h & \langle base(S') \Leftarrow S'\rangle^n\langle S' \Leftarrow base(S')\rangle^p\,(\phi(v)\,[\langle base(S) \Leftarrow S\rangle^p\, s_f]) \\ = & \phi([(v\; v'^{c,n,p})^{c',p,n}]) \end{array}$$

If S is a base refinement type and $s_f \to_h \texttt{blame}(n)$, then:

$$\begin{array}{ll} & \langle base(S') \Leftarrow S'\rangle^n\,[[\langle(S \to S') \Leftarrow base(S \to S')\rangle^p\,\phi(v)]\; s_f] \\ \to_h & \langle base(S') \Leftarrow S'\rangle^n\,[[\langle(S \to S') \Leftarrow base(S \to S')\rangle^p\,\phi(v)]\; \texttt{blame}(n)] \\ \to_h & \texttt{blame}(n) \\ & \phi([(v\; v'^{c,n,p})^{c',p,n}]) \\ = & \langle base(S') \Leftarrow S'\rangle^n[\langle S' \Leftarrow base(S')\rangle^p\,(\phi(v)\,[\langle base(S) \Leftarrow S\rangle^p\, s_f])] \\ \to_h & \langle base(S') \Leftarrow S'\rangle^n[\langle S' \Leftarrow base(S')\rangle^p\,(\phi(v)\,[\langle base(S) \Leftarrow S\rangle^p\, \texttt{blame}(n)])] \\ \to_h & \texttt{blame}(n) \end{array}$$

Otherwise, S is a base refinement type and $s_f \to_h \phi(v')$, and:

$$\begin{array}{ll} & \langle base(S') \Leftarrow S'\rangle^n\,[[\langle(S \to S') \Leftarrow base(S \to S')\rangle^p\,\phi(v)]\; s_f] \\ \to_h & \langle base(S') \Leftarrow S'\rangle^n\,[[\langle(S \to S') \Leftarrow base(S \to S')\rangle^p\,\phi(v)]\; \phi(v')] \\ \to_h & \langle base(S') \Leftarrow S'\rangle^n[\langle S' \Leftarrow base(S')\rangle^p\,(\phi(v)\,[\langle base(S) \Leftarrow S\rangle^p\; \phi(v')])] \\ & \phi([(v\; v'^{c,n,p})^{c',p,n}]) \\ = & \langle base(S') \Leftarrow S'\rangle^n[\langle S' \Leftarrow base(S')\rangle^p\,(\phi(v)\,[\langle base(S) \Leftarrow S\rangle^p\, s_f])] \\ \to_h & \langle base(S') \Leftarrow S'\rangle^n[\langle S' \Leftarrow base(S')\rangle^p\,(\phi(v)\,[\langle base(S) \Leftarrow S\rangle^p\, \phi(v')])] \end{array}$$

[E-CTX] Suppose $E[t_1] \to_c E[t_2]$ and $t_1 \to_c t_2$. This case holds by inspection. $\forall E.\phi(E[t_1]) \to_h \phi(E[t_2])$.

□

Lemma 10 *The operational semantics for λ^C and λ^H are deterministic.*

Restatement of Theorem 2 (Behavioral Equivalence) *If $t \to_c^* t'$ then $\exists s$ such that $\phi(t) \to_h^* s$ and $\phi(t') \to_h^* s$.*

PROOF: This proof is by induction on the length of the reduction $t \to_c^* t'$. The notation $\to_c^n$ and $\to_h^n$ indicates that n steps have occurred.
The base case where $n = 0$ is trivially true.
For the inductive case, suppose $t \to_c^n t' \to_c t''$.
By induction $\exists m_1, m_2 \in \mathbb{N}, s'$ such that $\phi(t) \to_h^{m_1} s'$ and $\phi(t') \to_h^{m_2} s'$.
By Lemma 9, $\exists m_3, m_4 \in \mathbb{N}, s''$ such that $\phi(t') \to_h^{m_3} s''$ and $\phi(t'') \to_h^{m_4} s''$.
Suppose $m_3 \leq m_2$. Then, since the evaluation from $\phi(t')$ is deterministic (Lemma 10), we have that $\phi(t) \to_h^{m_1} s'$ and $\phi(t'') \to_h^{m_4} s'' \to_h^{m_2-m_3} s'$.
The case where $m_3 \geq m_2$ is symmetric. □