1-17-2010

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

Michael Greenberg, Benjamin Pierce, and Stephanie Weirich. **Contracts Made Manifest**. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 353-364, Madrid, Spain, January 2010. ACM

doi:10.1145/1706299.1706341

Contracts Made Manifest

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

Our work extends that of Gronski and Flanagan, who defined a latent calculus lambdac and a manifest calculus lambdah, gave a translation phi from lambdac to lambdah, and proved that, if a lambdac term reduces to a constant, then so does its phiimage. We enrich their account with a translation psi from lambdah to lambdac 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 lambdah and two dialects ("lax" and "picky") of lambdac, establish type soundness-a substantial result in itself, for lambdah-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.

Disciplines
Computer Sciences

Comments
Michael Greenberg, Benjamin Pierce, and Stephanie Weirich. **Contracts Made Manifest**. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 353-364, Madrid, Spain, January 2010. ACM

doi:10.1145/1706299.1706341

Contracts Made Manifest

Michael Greenberg
University of Pennsylvania
Philadelphia, PA

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA

Stephanie Weirich
University of Pennsylvania
Philadelphia, PA

Abstract

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

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

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

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics
; D.2.4 [*Software Engineering*]: Software/Program Verification—Programming by contract

General Terms Languages, Theory

Keywords Contract, refinement type, dynamic checking, blame, precondition, postcondition, translation

1. Introduction

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

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

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

Findler and Felleisen's work sparked a resurgence of interest in contracts, and in the intervening years a bewildering variety of related systems have been studied. Broadly, these come in two different sorts. In systems with *latent* contracts, types and contracts are orthogonal features. Examples of this style include Findler and Felleisen's original system, Hinze et al. [2006], Blume and McAllester [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 work of Ou et al. [2004], Wadler and Findler [2009], and Gronski et al. [2006].

The key feature of manifest systems is that descriptions like $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\}$ are incorporated into the type system as *refinement types*. Values of refinement type are introduced via *casts* like $\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\}\rangle^{l}\,n$, which has static type $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\}$ and checks, dynamically, that n really is nonzero, raising $\Uparrow l$ otherwise. Similarly, $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\,x\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}\rangle^{l}\,n$ casts an integer that is statically known to be nonzero to one that is statically known to be positive.

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

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

where $I = \{x{:}\mathsf{Int} \mid \mathsf{true}\}$ and $P = \{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}$. The sequence of events when f is applied to some argument n (of type P) is similar to what we saw before: first, n is cast from P to I (it happens that 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 I to P, raising $\Uparrow l$ if this fails.

One point to note here is that casts have just one label, while contract checks in the latent system have two. This difference is not fundamental, but rather a question of the pragmatics of assigning responsibility: both latent and manifest systems can be given more or less rich algebras of blame. Informally, a function contract check

POPL'10, January 17–23, 2010, Madrid, Spain.

$\langle c_1 \mapsto c_2 \rangle^{l,l'} f$ divides responsibility for f's behavior between its body and its environment: the programmer is saying "If f is ever applied to an argument that doesn't pass c_1, I refuse responsibility ($\Uparrow l'$), whereas if f's result for good arguments doesn't satisfy c_2, I accept responsibility ($\Uparrow l$)." In a manifest system, the programmer who writes $\langle R_1 \to R_2 \Rightarrow S_1 \to S_2 \rangle^{l} f$ is saying "Although all I know statically about f is that its results satisfy R_2 when it is applied to arguments satisfying R_1, I assert that it's OK 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 seem intuitively to be much the same thing as typecasts in manifest systems, the formal correspondence is not immediate. This has led to some confusion in the community about the nature of contracts. Indeed, as we will see, matters become yet murkier in richer languages with features such as dependency.

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

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

Second, and more significantly, we extend the whole story to allow dependent function contracts in λ_{C} and dependent arrow types in λ_{H}. Dependency is extremely handy in contracts, as it allows for precise specifications of how the results of functions depend on their arguments. For example, here is a contract that we might use with an implementation of vector concatenation:

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

Adding dependent contracts to λ_{C} is easy: the dependency is all in the contracts and the types stay simple. We have just one significant design choice: should domain contracts be rechecked when the bound variable appears the codomain contract? This leads to two dialects of λ_{C}, one which does recheck (*picky* λ_{C}) and one which does not (*lax* λ_{C}). The choice is not clear—dependent contract systems have typically used the lax rule, while the picky one is arguably more correct—so we consider both. In λ_{H}, on the other hand, dependency significantly complicates the metatheory, requiring the addition of a denotational semantics for types and kinds to break a potential circularity in the definitions, plus an intricate sequence of technical lemmas involving parallel reduction to establish type soundness. (Although Gronski and Flanagan worked only with nondependent λ_{C} and λ_{H}, Knowles and Flanagan [2009] showed soundness for a variant of dependent λ_{H} in which order of evaluation is nondeterministic and failed casts get stuck instead of raising blame. See Section 7.)

Surprisingly, the tight correspondence between λ_{C} and λ_{H} breaks down in the dependent case: the natural generalization of the translations does not preserve blame exactly. Indeed, we can

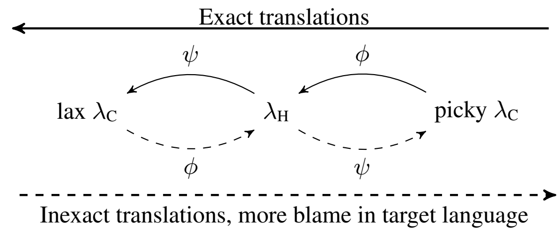

Figure 1. The axis of blame

place λ_{H} between the two variants of λ_{C} on an "axis of blame" (Figure 1), where behavior is preserved exactly when moving left on the axis (from picky λ_{C} to λ_{H} to lax λ_{C}), but translated terms can blame more than their pre-images when moving right.[1] The discrepancy arises in the case of "abusive" contracts, such as

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

where $I = \{x{:}\mathsf{Int} \mid \mathsf{true}\}$ and $N = \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}$. This rather strange contract has the form $f{:}c_1 \mapsto c_2$, where c_2 uses f in a way that violates c_1! In particular, if we apply it (in lax λ_{C}) to $\lambda f{:}\mathsf{Int} \to \mathsf{Int}.\ 0$ and then apply the result to $\lambda x{:}\mathsf{Int}.\ x$ and 5, the final result will be 5, since $\lambda x{:}\mathsf{Int}.\ x$ does satisfy the contract $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}$ and 5 satisfies the contract $\{z{:}\mathsf{Int} \mid (\lambda x{:}\mathsf{Int}.\ x)\, 0 \ = \ 0\}$. However, the translation of f into λ_{H} inserts an extra check, wrapping the occurrence of f in the codomain contract with a cast from $N \to I$ to $I \to I$, which fails when the wrapped function is applied to 0. We discuss this phenomenon in greater detail in Section 4.

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

A long version of the paper with definitions and proofs in full can be found at `http://www.cis.upenn.edu/~mgree/papers/contracts_tr.pdf`.

2. The nondependent languages

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

The language λ_{C}

The language λ_{C} is the simply typed lambda calculus straightforwardly augmented with contracts. The most interesting feature is the *contract* term $\langle c \rangle^{l,l'}$, which, when applied to a term t, dynamically ensures that t and its surrounding context satisfy c. If t doesn't satisfy c, then the *positive* label l will be blamed and the whole term will reduce to $\Uparrow l$; on the other hand, if the context doesn't treat $\langle c \rangle^{l,l'}\ t$ as c demands, then the negative label l' will be blamed and the term will reduce to $\Uparrow l'$. Contracts come in two forms: base contracts $\{x{:}B \mid t\}$ over a base type B and higher-

[1] There might, in principle, be some other way of defining ϕ and ψ that (a) preserves types, (b) maps base contracts to refinement-type casts and function contracts to arrow-type casts (and vice versa), and (c) induces an exact behavioral equivalence. After considering a number of alternatives, we conjecture that no such ϕ and ψ exist.

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

Figure 2. Base types and constants for λ_C and λ_H

Types and contracts

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

Terms, values, results, and evaluation contexts

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

Figure 3. Syntax for λ_C

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

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

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

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

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

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

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

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

Figure 4. Operational semantics for λ_C

order contracts $c_1 \mapsto c_2$, which check the arguments and results of functions.

The syntax of λ_C appears in Figure 3, with some common definitions (shared with λ_H) in Figure 2. Besides the contract term $\langle c\rangle^{l,l'}$, λ_C includes first-order constants k, blame, and *active checks* $\langle\{x{:}B \mid t_1\}, t_2, k\rangle^{l}$. Active checks do not appear in source programs; they are a technical artifact of the small-step operational semantics, as we explain below. Also, note that we only allow contracts over base types B: we have function contracts, like $\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \mapsto \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}$, but not contracts over functions, like $\{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\, \mathsf{true} = f\, \mathsf{false}\}$. We discuss this further in Section 8.

Values v include abstractions, contracts, function contracts applied to values, and constants; a *result* r is either a value or $\Uparrow l$ for some l. We define constants using three constructions: the set $\mathcal{K}_B$, which contains constants of base type B; the type-assignment function ty_c, which maps constants to first-order types of the form $B_1 \to B_2 \to \ldots \to B_n$ (and which is assumed to agree with $\mathcal{K}_B$); and the denotation function $[\![-]\!]$ which maps constants to functions

$$\boxed{\Gamma \vdash t : T} \qquad \frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T\_VAR}$$

$$\frac{}{\Gamma \vdash k : \mathrm{ty}_c(k)} \quad \text{T\_CONST}$$

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

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

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

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

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

$$\boxed{\vdash_c c : T} \qquad \frac{x{:}B \vdash t : \mathsf{Bool}}{\vdash_c \{x{:}B \mid t\} : B} \quad \text{T\_BASEC}$$

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

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

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

Figure 5. Typing rules for λ_C

from constants to constants (or blame, to allow for partiality). Denotations must agree with ty_c. We assume that Bool is among the base types, with $\mathcal{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

The operational semantics is given in Figure 4. It includes six rules for basic (small-step, call-by-value) reductions, plus two rules that involve evaluation contexts E (Figure 3). The evaluation contexts implement left-to-right evaluation for function application. If $\Uparrow l$ appears in the active position of an evaluation context, it is propagated to the top level. As usual, values (and results) do not step.

The first two basic rules are standard, implementing primitive reductions and β-reductions for abstractions. In these rules, arguments must be values v. Since constants are first-order, we know that when E_CONST applies to a well-typed application, the argument is not just a value, but a constant.

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

Types

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

Terms, values, results, and evaluation contexts

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

Figure 6. Syntax for λ_{H}

is responsible for the argument), and the result of the application is checked with the result contract.

The typing rules for λ_{C} (Figure 5) are mostly standard. We give types to constants using the type-assignment function $\mathrm{ty_c}$. Blame expressions have all types. Contracts are checked for well-formedness using the judgment $\vdash_c\ c : T$, comprising the rules T_BASEC, which requires that the checking term in a base contract return a boolean value when supplied with a term of the right type, and T_FUNC. Note that the predicate t in a contract $\{x{:}B \mid t\}$ can contain at most x free, since we are considering only nondependent contracts for now. Contract application, like function application, is checked using T_APP.

The T_CHECKING rule only applies in the empty context (active checks are only created at the top level during evaluation). The rule ensures that the contract $\{x{:}B \mid t_1\}$ has the right base type for the constant k, that the check expression t_2 has a boolean type, and that the check is actually checking the right contract. The latter condition is formalized by the T_IMP rule: $\vdash\ t_2 \supset t_1\{x := k\}$ asserts that if t_2 evaluates to true, then the original check $t_1\{x := k\}$ must also evaluate to true. This requirement is needed for two reasons: first, nonsensical terms like $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\,x\}, \mathsf{true}, 0\rangle^l$ should not be well typed; and second, we use this property in showing that the translations are type preserving (see Section 5). This rule obviously makes typechecking for the full "internal language" with checks undecidable, but excluding checks recovers decidability.

The language λ_{H}

Our second core calculus, nondependent λ_{H}, extends the simply typed lambda-calculus with *refinement types* and *cast expressions*. The syntax appears in Figure 6. Unlike λ_{C}, which separates contracts from types, λ_{H} combines them into refined base types $\{x{:}B \mid s_1\}$ and function types $S_1 \to S_2$. As for λ_{C}, we do not allow refinement types over functions, nor do we allow refinements of refinements. Unrefined base types B are *not* valid types; they must be wrapped in a trivial refinement, as the *raw* type $\{x{:}B \mid \mathsf{true}\}$. The terms of the language are mostly standard, including variables, the same first-order constants as λ_{C}, blame, abstractions, and applications. The cast expression $\langle S_1 \Rightarrow S_2\rangle^l$ dynamically checks that a term of type S_1 can be given type S_2. Like λ_{C}, active checks are used to give a small-step semantics to cast expressions.

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

The small-step, call-by-value semantics in Figure 7 comprises six basic rules and two rules involving evaluation contexts F. Each rule corresponds closely to its counterpart in λ_{C}.

$$\frac{}{k\ w \ \longrightarrow_h\ [\![k]\!](w)} \quad \text{F\_CONST}$$

$$\frac{}{(\lambda x{:}S_1.\, s_2)\ w_2 \ \longrightarrow_h\ s_2\{x := w_2\}} \quad \text{F\_BETA}$$

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

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

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

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

$$\frac{s_1 \ \longrightarrow_h\ s_2}{F\,[s_1] \ \longrightarrow_h\ F\,[s_2]} \quad \text{F\_COMPAT}$$

$$\frac{}{F\,[\Uparrow l] \ \longrightarrow_h\ \Uparrow l} \quad \text{F\_BLAME}$$

Figure 7. Operational semantics for λ_{H}

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

The typing rules for λ_{H} (Figure 8) are also similar to those of λ_{C}. Just as the λ_{C} rule T_CONTRACT checks to make sure that the contract has the right form, the λ_{H} rule S_CAST ensures that the two types in a cast are well-formed and have the same simple-type skeleton, defined as $\lfloor - \rfloor : S \to T$ (pronounced "erase S"):

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

We define a similar operator, $\lceil - \rceil : S \to S$ (pronounced "raw" S), which trivializes all refinements:

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

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

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

An important difference is that λ_{H} has subtyping. The S_SUB rule allows an expression to be promoted to any well-formed supertype. Refinement types are supertypes if, for all constants of

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

$$\frac{x{:}S \in \Delta}{\Delta \vdash x \,:\, S} \qquad \text{S\_VAR}$$

$$\frac{}{\Delta \vdash k \,:\, \mathrm{ty_h}(k)} \qquad \text{S\_CONST}$$

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

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

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

$$\frac{\vdash S}{\Delta \vdash \Uparrow l \,:\, S} \qquad \text{S\_BLAME}$$

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

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

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

$$\frac{\forall k \in \mathcal{K}_B.\ \vdash s_1\{x := k\} \supset s_2\{x := k\}}{\vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}} \qquad \text{SSUB\_REFINE}$$

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

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

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

$\boxed{\vdash S}$

$$\frac{}{\vdash \{x{:}B \mid \mathsf{true}\}} \qquad \text{SWF\_RAW}$$

$$\frac{x{:}\{x{:}B \mid \mathsf{true}\} \vdash s \,:\, \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\vdash \{x{:}B \mid s\}} \qquad \text{SWF\_REFINE}$$

$$\frac{\vdash S_1 \qquad \vdash S_2}{\vdash S_1 \to S_2} \qquad \text{SWF\_FUN}$$

Figure 8. Typing rules for λ_{H}

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 this makes type checking undecidable[2]; subtyping is just a technical device for ensuring type preservation. Consider the following reduction:

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

The source term is well-typed at $\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$. Since it evaluates to 1, we would like to have $\Delta \vdash 1 \,:\, \{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\}$. To have type preservation in general, though, $\mathrm{ty_h}(1)$ must be a subtype of $\{x{:}\mathsf{Int} \mid s\}$ *whenever* $s\{x := 1\} \longrightarrow_h^* \mathsf{true}$. That is, constants of base type must have "most-specific" types. One way to satisfy this requirement is to set $\mathrm{ty_h}(k) = \{x{:}B \mid x = k\}$ for $k \in \mathcal{K}_B$; then if $s\{x := k\} \longrightarrow_h^* \mathsf{true}$, we have $\vdash \mathrm{ty_h}(k) <: \{x{:}B \mid s\}$.

Standard progress and preservation theorems hold for λ_{H}. We can also obtain a semantic type soundness theorem as a restriction of the one for dependent λ_{H} (Theorem 4.2).

3. The nondependent translations

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

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.

For ψ, translating from λ_{H}'s rich types to λ_{C}'s simple types is easy: we just erase the types to their simple skeletons. The interesting case is translating the cast $\langle S_1 \Rightarrow S_2 \rangle^{l}$ to a contract by translating the pair of types together, $\langle \psi(S_1, S_2) \rangle^{l,l}$. So ψ translates λ_{H} terms to λ_{C} terms and *pairs* of λ_{H} types to λ_{C} contracts:

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

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

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

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

Translating from λ_{C} to λ_{H}, we are moving from a simple type system to a rich one. The translation ϕ (essentially the same as Gronski and Flanagan's) generates terms in λ_{H} with *raw* types—λ_{H} types with trivial refinements, corresponding to λ_{C}'s simple types.

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

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

where the translation of contracts to refined types is:

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

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

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

[2] Flanagan [2006] and Knowles and Flanagan [2009] discuss trade-offs between static and dynamic checking that allow for decidable type systems and subtyping.

translates to the λ_{H} term

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

The domain of the negative cast checks that f's argument is nonzero with $\langle\lceil \mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}\rangle^{l'}$. The domain of the positive cast does nothing, since $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\} \Rightarrow \lceil \mathsf{Int}\rceil\rangle^{l}$ has no effect. Similarly, the codomain of the negative cast does nothing while the codomain of the positive cast checks that the result is positive. Separating the checks allows λ_{H} to keep track of blame labels, mimicking λ_{C}. This embodies the idea of contracts as pairs of projections [Findler 2006]. Note that bulletproofing is "overkill" at base type: for example, $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\, x\}\rangle^{l,l'}$ translates to

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

Only the positive cast does anything—the negative cast into $\lceil \mathsf{Int}\rceil$ always succeeds. This asymmetry is consistent with λ_{C}, where base-type contracts also ignore the negative label.

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

4. The dependent languages

We now extend λ_{C} to dependent function contracts and λ_{H} to dependent functions. The changes are summarized in Figure 9 (for λ_{C}) and Figures 10 and 11 (for λ_{H}). Very little needs to be changed in λ_{C}, since contracts and types barely interact; the changes to E_CDECOMP and T_FUNC are the important ones. Adding dependency to λ_{H} is more involved. In particular, adding contexts to the subtyping judgment entails adding contexts to S_IMP. To avoid a dangerous circularity, we define closing substitutions in terms of a separate type semantics. Additionally, the new F_CDECOMP rule has a slightly tricky (but necessary) asymmetry, explained below.

Dependent λ_{C}

Dependent λ_{C} has been studied since Findler 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 ϕ easier.

The new T_REFINEC, T_FUNC, and E_CDECOMP rules in Figure 9 suffice to add dependency to λ_{C}. To help us work with the translations, we also make some small changes to the bindings in contexts, tracking the labels on a contract check throughout the contract well-formedness judgment. Note that T_FUNC adds $x{:}{c_1}^{l',l}$ to the context when checking the codomain of a function contract, swapping blame labels. We add a new variable rule, T_VARC, that treats $x{:}c^{l,l'}$ as if it were its skeleton, $x{:}\lfloor c\rfloor$. While unnecessary for λ_{C}, this new binding form helps ϕ preserve types. (See Section 6.1).

Two different variants of the E_CDECOMP rule can be found in the literature: we call them *lax* and *picky*. The original rule in Findler and Felleisen [2002] is lax (like most other contract

Contracts and contexts

$$c \quad ::= \quad \{x{:}B \mid t\} \ \mid\ x{:}c_1 \mapsto c_2$$

$$\Gamma \quad ::= \quad \emptyset \ \mid\ \Gamma, x{:}T \ \mid\ \Gamma, x{:}c^{l,l'}$$

Operational Semantics

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

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

Typing rules

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_VART} \qquad \frac{x{:}c^{l,l'} \in \Gamma}{\Gamma \vdash x : \lfloor c\rfloor}\ \text{T\_VARC}$$

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

$$\frac{\Gamma, x{:}B \vdash t : \mathsf{Bool}}{\Gamma \vdash_c^{l,l'} \{x{:}B \mid t\} : B}\ \text{T\_REFINEC}$$

$$\frac{\Gamma \vdash_c^{l',l} c_1 : T_1 \qquad \Gamma, x{:}{c_1}^{l',l} \vdash_c^{l,l'} c_2 : T_2}{\Gamma \vdash_c^{l,l'} x{:}c_1 \mapsto c_2 : T_1 \to T_2}\ \text{T\_FUNC}$$

Figure 9. Changes for dependent λ_{C}

calculi): it does not recheck c_1 when substituting v' into c_2. Hinze et al. [2006] choose instead to be picky, substituting $\langle c_1\rangle^{l',l}\ v'$ into c_2 because it makes their conjunction contract idempotent. We can show (straightforwardly) that both enjoy standard progress and preservation properties. Below, we consider translations to and from both dialects of λ_{C}: picky λ_{C} using only E_CDECOMPPICKY in Sections 5.1 and 6.2, and lax λ_{C} using only E_CDECOMPLAX in Sections 5.2 and 6.1.

Dependent λ_{H}

Now we come to the challenging part: dependent λ_{H} and its proof of type soundness.[3] These results require the most complex metatheory in the paper, because we need some strong properties about call-by-value evaluation. (The benefit of a CBV semantics is a better treatment of blame. By contrast, Knowles and Flanagan [2009] cannot treat failed casts as exceptions because that would destroy confluence. They treat them as stuck terms.) The needed extensions are detailed in Figures 10 and 11.[4]

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

[4] The semantics in these figures is the same as that of Knowles and Flanagan [2009] except for the evaluation relation, the treatment of blame, and a change to the type semantics that we discuss below.

Types

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

Operational semantics

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

Typing rules

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

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

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

$$\frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 \supset s_2}{\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}} \quad \text{SSUB\_REFINE}$$

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

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

Closing substitutions

$$\frac{}{\emptyset \models \emptyset} \ \text{SCS\_EMPTY} \qquad \frac{s \in [\![S]\!] \qquad \Delta\{x := s\} \models \sigma}{x{:}S, \Delta \models \sigma\{x := s\}} \quad \text{SCS\_EXT}$$

Figure 10. Changes for dependent λ_H

First, we enrich the type system with dependent function types, $x{:}S_1 \to S_2$, where x may appear in S_2. A new application rule, S_APP, substitutes the argument into the result type of the application. We generalize SWF_REFINE to allow refinement-type predicates that use variables from the enclosing context. SWF_FUN adds the bound variable to the context when checking the codomain of function types. In SSUB_FUN, subtyping for dependent function types remains contravariant, but we also add the argument variable to the context with the smaller type.

We need to be careful when implementing higher-order dependent casts in the rule F_CDECOMP. As the cast decomposes, the variables in the codomain types of such a cast must be replaced. However, this substitution is asymmetric; on one side, we cast the argument and on the other we do not. This behavior is required for type soundness. For suppose we have $\Delta \vdash x{:}S_{11} \to S_{12}$ and $\Delta \vdash x{:}S_{21} \to S_{22}$ with equal skeletons, and values $\Delta \vdash w : (x{:}S_{11} \to S_{12})$ and $\Delta \vdash w' : S_{21}$. Then $\Delta \vdash (\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\, w)\, w' : S_{22}\{x := w'\}$. When we decompose the cast, we must make *some* substitution into S_{12} and S_{22}, but which? It is clear that we must substitute w' into S_{22}, since the original application has type $S_{22}\{x := w'\}$. Decomposing the cast will produce the inner application $\Delta \vdash w\,(\langle S_{21} \Rightarrow S_{11}\rangle^l\, w') : S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\, w'\}$. In order to apply the codomain cast, we must substitute $\langle S_{21} \Rightarrow S_{11}\rangle^l\, w'$ into S_{12}. This calculation determines the form of F_CDECOMP.

Denotations of types and kinds

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

Semantic judgments

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

Figure 11. Type and kind semantics for dependent λ_H

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

Care is needed here to prevent the typing rules from becoming circular: the typing rule S_SUB references the subtyping judgment, the subtyping rule SSUB_REFINE references the implication judgment, and the single implication rule S_IMP has $\Delta \models \sigma$ in a negative position. To avoid circularity, $\Delta \models \sigma$ must not refer back to the other judgments.

We can achieve this by building the syntactic rules on top of a denotational semantics for λ_H's types.[5] The idea is that the semantics of a type is a set of closed terms that is defined independently of the syntactic typing relation, but that turns out to contain all closed well-typed terms of that type. Thus, in the definition of $\Delta \models \sigma$, we quantify over a somewhat larger set than 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 11. 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 go to well-typed results. By construction, these sets include only terminating terms that do not get stuck.

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

The main things we want to know about the type semantics is *semantic type soundness*: if $\emptyset \vdash s : S$, then $s \in [\![S]\!]$. However, to prove this, we must generalize it. In the bottom of Figure 11, we define three *semantic judgements* that correspond to each of the three typing judgments. (Note that the third one requires the definition of a *kind* semantics that picks out well-behaved types—those whose embedded terms belong to the type semantics.) We then show that the typing judgments imply their semantic counterparts.

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

Result correspondence

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

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

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

Term correspondence

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

Figure 12. A blame-exact result/term correspondence

Contract / type correspondence

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

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

Dual closing substitutions

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

Figure 13. Blame-exact correspondence for ϕ from picky λ_C

4.2 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 $\Delta \models S$.

The first part follows by induction on the subtyping judgment. However, we run into some complications with the second and third parts (which must be proven together). The crux of the difficulty lies with the S_APP rule. Suppose the application $s_1\, s_2$ was well typed and $s_1 \in [\![x{:}S_1 \to S_2]\!]$ and $s_2 \in [\![S_1]\!]$. According to S_APP, the application's type is $S_2\{x := s_2\}$. By the type semantics defined in Figure 11, if $s_1 \in [\![x{:}S_1 \to S_2]\!]$, then $s_1\, q \in [\![S_2\{x := q\}]\!]$ for any $q \in [\![S_1]\!]$. Sadly, s_2 isn't necessarily a result! We do know, however, that $s_2 \in [\![S_1]\!]$, so $s_2 \longrightarrow_h^* q_2$ by strong normalization (Lemma 4.1). We need to ask, then, how the type semantics of $S_2\{x := s_2\}$ and $S_2\{x := q_2\}$ relate.

We can show that the two type semantics are in fact equal using a parallel reduction technique. We define a parallel reduction relation $\Rrightarrow$ on terms and types that allows redices in different parts of a term (or type) to be reduced in the same step, and we prove that types that parallel-reduce to each other—like $S_2\{x := s_2\}$ and $S_2\{x := q_2\}$—have the same semantics (see the long version for details). The definition of parallel reduction is standard, though we need to be careful to make it respect our call-by-value reduction order: the β-redex $(\lambda x{:}S_1.\, s_1)\, s_2$ should not be contracted unless s_2 is a value, since doing so can change the order of effects. (Other redices within s_1 and s_2 can safely reduce.) The proof requires a longish sequence of technical lemmas that essentially show that $\Rrightarrow$ commutes with $\longrightarrow_h^*$. Since the proofs require fussy symbol manipulation, we've done these proofs in Coq. Our development is available at `http://www.cis.upenn.edu/~mgree/papers/lambdah_parred.tgz`.

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

Theorem 4.2 gives us type soundness, and it combines with Lemma 4.1 for an even stronger result: well-typed programs always evaluate to values of appropriate (semantic) type.

5. Exact translations

Translations moving left on the axis of blame—from picky λ_C to λ_H, and from λ_H to lax λ_C—are exact. That is, we can show a tight behavioral correspondence between terms and their translations (see Figure 12). We read $t \sim s : T$ as "t corresponds with s at type T". Terms corresponding at B both go to $k \in \mathcal{K}_B$ or to $\Uparrow l$.

5.1 Translating picky λ_C to λ_H: dependent ϕ

The full definition of ϕ is in Figure 14. One point to note is that, in the dependent case, we need to translate *derivations* of well-formedness and well-typing of λ_C contexts, terms, and contracts into λ_H contexts, terms, and types. We translate derivations to ensure type preservation, translating T_VART and T_VARC derivations differently: we leave variables of simple type alone, but we cast variables bound to contracts.

To see why we need this distinction, consider the function contract $f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\, 0 = 0\}$. Note that this contract is well-formed in λ_C, but that the codomain "abuses" the bound variable. A naive translation will *not* be well-typed in λ_H: $f\, 0$ will not be typeable when f has type $x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \to \lceil \mathsf{Int} \rceil$, since f only accepts positive arguments. The problem is that SWF_FUN can add a (possibly refined) type to the context when checking the codomain, so we need to restore the "variables have raw types" invariant. By tracking which variables were bound by contracts in λ_C, we can be sure to cast them to raw types when they're referenced. We therefore translate the contract above to $f{:}S \to \{z{:}\mathsf{Int} \mid (\langle S \Rightarrow \lceil \mathsf{Int} \to \mathsf{Int} \rceil\rangle^{l'} f)\, 0 = 0\}$, where $S = x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\, x\} \to \lceil \mathsf{Int} \rceil$. This (partially) motivates the $x{:}c^{l,l'}$ binding form in dependent λ_C.

Constants translate to themselves. One technical point: to maintain the raw type invariant, we need λ_H's higher-order constants to have typings that can be seen as raw by the subtyping relation, i.e., $\Delta \vdash \mathrm{ty_h}(k) <: \lceil \mathrm{ty_c}(k) \rceil$. This slightly restricts the types we might assign to our constants, e.g., we cannot say $\mathrm{ty_h}(\mathsf{sqrt}) = x{:}\{x{:}\mathsf{Float} \mid x \geq 0\} \to \{y{:}\mathsf{Float} \mid (y * y) = x\}$, since it is not the case that $\Delta \vdash \mathrm{ty_h}(\mathsf{sqrt}) <: \lceil \mathsf{Float} \to \mathsf{Float} \rceil$. Since its domain cannot be refined, $[\![\mathsf{sqrt}]\!]$ must be defined for all $k \in \mathcal{K}_{\mathsf{Float}}$, e.g., $[\![\mathsf{sqrt}]\!](-1)$ must be defined. We've already required that denotations be total over their simple types in λ_C, and λ_H uses the same denotation function $[\![-]\!]$, so this requirement does not seem too severe. We could instead translate k to $\langle \mathrm{ty_h}(k) \Rightarrow \lceil \mathrm{ty_h}(k) \rceil\rangle^{l_0}\, k$; however, in this case the nondependent fragments of the languages would no longer correspond exactly.

We extend the term correspondence of Figure 12 to contracts and types, lifting the correspondences to open terms using dual closing substitutions. For a binding $x{:}c^{l,l'} \in \Gamma$, we use ϕ to insert the negative cast (labelled with l') and closing substitutions (in Figure 13) to insert the positive cast (labelled with l). Do not be confused by the label used for function contract correspondence—this definition does, in fact, match up with closing substitutions. A binding $x{:}c^{l,l'} \in \Gamma$ must have come from the domain of an application of T_FUNC, so the labels on the binding are *already* swapped when ϕ or $\Gamma \models \delta$ sees them. In the definition of function contract correspondence, we swap manually—whence the l' on the

Terms

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

Types

$$
\begin{array}{rcl}
\phi(\Gamma \vdash_c^{l,l'} \{x{:}B \mid t\} : B) & = & \{x{:}B \mid \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})\} \\
\phi(\Gamma \vdash_c^{l,l'} x{:}c_1 \mapsto c_2 : T_1 \to T_2) & = & x{:}\phi(\Gamma \vdash_c^{l',l} c_1 : T_1) \to \phi(\Gamma, x{:}c_1{}^{l',l} \vdash_c^{l,l'} c_2 : T_2)
\end{array}
$$

Contexts

$$
\begin{array}{rcl}
\phi(\vdash \emptyset) & = & \emptyset \\
\phi(\vdash \Gamma, x{:}T) & = & \phi(\vdash \Gamma), x{:}\lceil T \rceil \\
\phi(\vdash \Gamma, x{:}c^{l,l'}) & = & \phi(\vdash \Gamma), x{:}\phi(\Gamma \vdash_c^{l,l'} c : \lfloor c \rfloor)
\end{array}
$$

Figure 14. The translation ϕ from dependent λ_{C} to dependent λ_{H}

Term translation

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

Type-to-contract translation

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

Figure 15. ψ mapping dependent λ_{H} to dependent λ_{C}

inserted cast. It helps to think of polarity in terms of position rather than the presence or absence of a prime.

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

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

We can now prove that ϕ preserves types, using Theorem 5.1 to show that ϕ preserves the implication judgment.

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

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

5.2 Translating λ_{H} to lax λ_{C}: dependent ψ

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

The full definition of ψ is in Figure 15. Most terms are translated homomorphically. In abstractions, the annotation is translated by erasing the refined λ_{H} type to its skeleton. As we mentioned in Section 3, the trickiest part is the translation of casts between function types: when generating the codomain contract from a cast between two function types, we perform the same asymmetric substitution as F_CDECOMP. Since ψ inserts new casts, we need to pick a blame label: $\psi(\langle S_1 \Rightarrow S_2 \rangle^l)$ passes l as an index to $\psi^l(S_1, S_2)$.

We reuse the term correspondence $t \sim s : T$ (Figure 12) and define a new contract/cast correspondence $c \sim S_1 \Rightarrow^l S_2 : T$ (Figure 16), relating contracts and pairs of λ_{H} types. This correspondence uses the term correspondence in the base type case and follows the pattern of F_CDECOMP in the function case. Since it inserts a cast in the function case, we index the relation with a label, just like ψ. We define closing substitutions ignoring the contracts in the context; we lift the relation to open terms in the standard way.

Contract / type correspondence

$$
\begin{array}{c}
\{x{:}B \mid t\} \sim \{x{:}B \mid s_1\} \Rightarrow^l \{x{:}B \mid s_2\} : B \iff \\
\forall k \in \mathcal{K}_B.\ t\{x := k\} \sim s_2\{x := k\} : \mathsf{Bool} \\[1em]
x{:}c_1 \mapsto c_2 \sim x{:}S_{11} \to S_{12} \Rightarrow^l S_{21} \to S_{22} : T_1 \to T_2 \iff \\
c_1 \sim S_{21} \Rightarrow^l S_{11} : T_1 \quad \wedge \\
\forall t \sim s : T_1. \\
c_2\{x := t\} \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\, s\} \Rightarrow^l S_{22}\{x := s\} : T_2
\end{array}
$$

Figure 16. Blame-exact correspondence for ψ into lax λ_{C}

We first show that s and $\psi(s)$ behaviorally correspond, and then we can prove that ψ is type preserving, using the behavioral correspondence to show that ψ preserves implication.

5.3 Theorem [Behavioral correspondence]:

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

5.4 Theorem [Type preservation for ψ]:

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

6. Inexact translations

The same translations ϕ and ψ can be used to move right on the axis of blame (Figure 1). However, in this direction the images of these translations blame strictly more than their pre-images.

6.1 Translating lax λ_{C} to λ_{H}

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

We can show the behavioral correspondence using a blame-inexact logical relation, defined in Figure 17. The behavioral corre-

Value correspondence

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

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

Term correspondence

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

Contract / type correspondence

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

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

Dual closing substitutions

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

Figure 17. Blame-inexact correspondence for ϕ from lax λ_{C}

spondence here, though weaker than before, is still pretty strong: if $t \sim_{\succ} s : B$ (read "t blames no more than s at type B"), then either $s \longrightarrow_h^* \Uparrow l$ or t and s both go to $k \in \mathcal{K}_B$. This correspondence differs slightly in construction from the earlier exact one—we define $\approx_{\succ}$ as a relation on *values*, while $\approx$ is a relation on *results*. Doing so simplifies our inexact treatment of blame. We again use the term correspondence to relate contracts and λ_{H} types. We then lift the correspondences to open terms (Figure 17). Closing substitutions map variables to corresponding terms of appropriate type. Note that closing substitutions ignore the contract part of $x{:}c^{l,l'}$ bindings, treating them as if they were $x{:}\lfloor c \rfloor$.

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

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

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

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

We can also show type preservation for terms not containing active checks. (We don't know that translated active checks are well typed, because Theorem 6.2 isn't strong enough to preserve the implication judgment. We only expect these checks to occur at runtime, so this is good enough: ϕ preserves the types of source programs.)

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

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

To see that the ϕ in Figure 14 does not give us exact blame, let us look at two examples; in both cases, a lax λ_{C} term goes to a value

Contract / type correspondence

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

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

Dual closing substitutions

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

Figure 18. Blame-inexact correspondence for ψ into picky λ_{C}

while its translation goes to blame. In the first example, blame is raised in λ_{H} due to bulletproofing. In the second, blame is raised due to the extra cast from the translation of T_VARC. For the first, let

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

We find $\langle c \rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_c^* 0$ but $(\lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^l\, (\langle \lceil S \rceil \Rightarrow S \rangle^l\, x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l$. For the second, let

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

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

6.2 Translating λ_{H} to picky λ_{C}

Terms in λ_{H} and their ψ-images in lax λ_{C} correspond exactly, as shown Section 5.2. When we change the operational semantics of λ_{C} to be *picky*, however, $\psi(s)$ blames (strictly) more often than s. Nevertheless, we can show an inexact correspondence, as we did for ϕ and lax λ_{C} in Section 6.1. We use a logical relation $\sim_{\prec}$ similar to $\sim_{\succ}$, used for ϕ into lax λ_{C} (Figure 17), though we reverse the asymmetry, allowing picky λ_{C} to blame more than λ_{H}. The proof follows the same general pattern: we first show that it is safe to add extra contract checks, then the correspondence for well-typed terms. We can also show type preservation for source programs (excluding active checks).

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

6.5 Theorem [Behavioral correspondence]:

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

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

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

Here is an example where a $\lambda_{\rm H}$ term reduces to a value while its ψ-image in picky $\lambda_{\rm C}$ term reduces to blame:

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

Let $w \;=\; (\lambda f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\, y\}).\ 0)$ and $w' \;=\; (\lambda x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\}.\ 0)$. On the one hand $(\langle S_1 \Rightarrow S_2\rangle^l\, w)\, w' \;\longrightarrow_h^*\; 0$, while $(\langle c\rangle^{l,l}\, \lambda f{:}\mathsf{Int}.\ 0)\, \lambda x{:}\mathsf{Int}.\ 0 \;\longrightarrow_c^*\; \Uparrow l$. This means we cannot hope to use ψ as an exact correspondence between $\lambda_{\rm H}$ and picky $\lambda_{\rm C}$. (Removing the extra cast ψ inserts into S_{12} doesn't affect our example, since ψ ignores S_{12} here.)

6.3 Restricted calculi

While ϕ from lax $\lambda_{\rm C}$ and ψ to picky $\lambda_{\rm C}$ don't induce exact behavioral correspondences in the dependent case, some useful restrictions of the languages *are* equivalent.

Gronski and Flanagan [2007] have already shown that ϕ induces an exact correspondence on the nondependent restriction. Since the lax/picky distinction requires dependency, exact equivalence in the nondependent case is a restriction of the results of Section 5.

Moreover, the first-order dependent restrictions of $\lambda_{\rm C}$ and $\lambda_{\rm H}$ also correspond exactly. The intuition here is that rechecking a first-order contract in a new context can't change the result of checking—first-order contracts can't be abusive. We can show this for ϕ using our existing parallel reduction for $\lambda_{\rm H}$. We can show it for ψ, as well, using a similar parallel reduction for $\lambda_{\rm C}$. For this second proof we assume (but do not prove) that evaluation and reduction in $\lambda_{\rm C}$ commute as they do in $\lambda_{\rm H}$.

7. Related work

Conferences in recent years have seen a profusion of papers on higher-order contracts and related features. This is all to the good, but for newcomers to the area it can be a bit overwhelming, especially given the great variety of technical approaches. To help reduce the level of confusion, in Figure 19 we summarize the important points of comparison between a number of systems that are closely related to ours.

The largest difference is between latent and manifest treatments of contracts—i.e., whether contract checking (under whatever name) is a completely dynamic matter or whether it leaves a "trace" that the type system can track.

Another major distinction (labeled "dep" in the figure) is the presence of dependent contracts or, in manifest systems, dependent function types. Latent systems with dependent contracts also vary in whether their semantics is lax or picky.

Next, most contract calculi use a standard call-by-value order of evaluation ("eval order" in the figure). Notable exceptions include those of Hinze et al. [2006], which is embedded in Haskell, Flanagan [2006], which uses a variant of call-by-name, and Knowles and Flanagan [2009], which uses full β-reduction (more on this below).

Another point of variation ("blame" in the figure) is how contract violations or cast failures are reported—by raising an exception or by getting stuck. We also return to this below.

The next two rows in the table ("checking" and "typing") concern more technical points in the papers most closely related to ours. In both Gronski and Flanagan [2007] and Flanagan [2006], the operational semantics checks casts "all in one go":

$$\frac{s_2\{x := k\} \;\longrightarrow_h^*\; \mathsf{true}}{\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\; k \;\longrightarrow_h\; k}$$

Such rules are formally awkward, and in any case they violate the spirit of a small-step semantics. Also, the formal definitions of $\lambda_{\rm H}$ in both Gronski and Flanagan [2007] and Flanagan [2006] involve a circularity between the typing, subtyping, and implication relations. Knowles and Flanagan [2009] improve the technical presentation of $\lambda_{\rm H}$ in both respects. In particular, they avoid circularity (as we do) by introducing a denotational interpretation of types and maintain small-step evaluation by using a new syntactic form of "partially evaluated casts" (like most of the other systems).

The main contributions of the present paper are (1) the dependent translations ϕ and ψ and their properties, and (2) the formulation and metatheory of dependent $\lambda_{\rm H}$. (Dependent $\lambda_{\rm C}$ is not a contribution on its own: many similar systems have been studied, and in any case its properties are simple.) The nondependent part of our ϕ translation essentially coincides with the one studied by Gronski and Flanagan [2007], and our behavioral correspondence theorem is essentially the same as theirs. Our ψ translation completes their story for the nondependent case, establishing a tight connection between the systems. The full dependent forms of ϕ and ψ studied here are novel, as is the observation that the correspondence between the latent and manifest worlds is more problematic in this setting.

Our formulation of $\lambda_{\rm H}$ is most comparable to that of Knowles and Flanagan [2009], but there are some significant differences. First, our cast-checking constructs are equipped with labels, and failed casts go to explicit blame—i.e., they raise labeled exceptions. In the $\lambda_{\rm H}$ of Knowles and Flanagan (though not the earlier one of Gronski and Flanagan), failed casts are simply stuck terms—their progress theorem says "If a well-typed term cannot step, then either it is a value or it contains a stuck cast." Second, their operational semantics uses full, non-deterministic β-reduction, rather than specifying a particular order of reduction, as we have done. This significantly simplifies parts of the metatheory by allowing them to avoid introducing parallel reduction. We prefer standard call-by-value reduction because we consider blame as an exception—a computational effect—and we want to be able to reason about *which* blame will be raised by expressions involving many casts.

The system studied by Ou et al. [2004] is also close in spirit to our $\lambda_{\rm H}$. The main difference is that, because their system includes general recursion, they restrict the terms that can appear in contracts to just applications involving predefined constants: only "pure" terms can be substituted into types, and these do not include lambda-abstractions. Our system (like all of the others in Figure 19—see the row labeled "any con") allows arbitrary user-defined boolean functions to be used as contracts.

Our description of $\lambda_{\rm C}$ is ultimately based on $\lambda_{\rm CON}$ [Findler and Felleisen 2002], though our presentation is slightly different in its use of checks. Hinze et al. [2006] adapted Findler and Felleisen-style contracts to a location-passing implementation in Haskell, using a picky dependent function contract rule.

Our $\lambda_{\rm H}$ type semantics in Section 4 is effectively a semantics of contracts. Blume and McAllester [2006] offers a semantics of contracts that is slightly different—our semantics includes blame at every type, while theirs explicitly excludes it. Xu et al. [2009] is also similar, though their "contracts" have no dynamic semantics at all: they are simply specifications.

We have discussed only a small sample of the many papers on contracts and related ideas. We refer the reader to Knowles and Flanagan [2009] for a more comprehensive survey. Another useful

	latent systems					manifest systems					
	FF02 (1)	HJL06 (2)	GF07 λ_C (3)	BM06 (4)	our λ_C	GF07 λ_H (3)	F06 (5)	KF09 (6)	WF09 (7)	OTMW04 (8)	our λ_H
dep (9)	✓ lax	✓ picky	$\times$	(10)	✓ either	$\times$	✓	✓	$\times$	✓	✓
eval order	CBV	lazy	CBV	CBV	CBV	CBV	CBN(11)	full β	CBV	CBV	CBV
blame (12)	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$ or $\bot$	$\Uparrow l$	$\Uparrow l$	stuck	stuck	$\Uparrow l$	$\Uparrow$	$\Uparrow l$
checking (13)	if	if	$\bigcirc$	active	active	$\bigcirc$	$\bigcirc$	active	active	if	active
typing (14)	✓	✓	✓	n/a	✓	$\times$	$\times$	✓	✓	✓	✓
any con (15)	✓	✓	✓	✓	✓	✓	✓	✓	✓	$\times$	✓

(1) Findler and Felleisen [2002]. (2) Hinze et al. [2006]. (3) Gronski and Flanagan [2007]. (4) Blume and McAllester [2006]. (5) Flanagan [2006]. (6) Knowles and Flanagan [2009]. (7) Wadler and Findler [2009]. (8) Ou et al. [2004]. (9) Does the system include dependent contracts or function types (✓) or not ($\times$) and, for latent systems, is the semantics lax or picky? (10) An "unusual" form of dependency, where negative blame in the codomain results in nontermination. (11) A nondeterministic variant of CBN. (12) Do failed contracts raise labeled blame ($\Uparrow l$), raise blame without a label ($\Uparrow$), get stuck, or sometimes raise blame and sometimes diverge ($\bot$)? (13) Is contract or cast checking performed using an "active check" syntactic form (active), an "if" construct with a refined typing rule (if), or "inlined" by making the operational semantics refer to its own reflexive and transitive closure ($\bigcirc$)? (14) Is the typing relation well defined (i.e., for dependently typed systems, is it based on a type semantics or, as in WF09, a "tagging" scheme), or is the definition circular? (15) Are arbitrary user-defined boolean functions allowed as contracts or refinements (✓), or only built-in ones ($\times$)?

Figure 19. Comparison of related systems

resource is Wadler and Findler [2007] (technically superceded by Wadler and Findler [2009], but with a longer related work section), which surveys work combining contracts with type Dynamic and related features.

There are also *many* other systems that 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 [Gronski et al. 2006] and Knowles and Flanagan [2009] both support mixed static and dynamic checking of contracts, using, e.g., a theorem prover. We have not addressed this aspect of their work, since we have chosen to work directly with the core calculus λ_H, which for them was the target of an elaboration function.

8. Future work

Our calculi are strongly normalizing; extending our results to systems that allow recursion is a natural next step. The changes seem nontrivial: with nontermination, inexact correspondences must allow not only more blame, but also more nontermination—each extra check is another opportunity for divergence.

Most studies of contracts, including ours, only allow refinements of base types; however, Blume and McAllester [2006] and Xu et al. [2009] also allow refinements of functions. This extension seems needed if contracts are to be combined with polymorphism, since in this setting we may want to refine type variables, which may later be substituted with types involving functions. We conjecture that dependent λ_H with function refinements is sound, but it is not clear how the translations will need to be modified.

Acknowledgments

Sewell and Zappa Nardelli's OTT tool was invaluable for organizing our definitions. We used Aydemir and Weirich's LNGen tool for the Coq development of parallel reduction. Brian Aydemir, João Belo, Chris Casinghino, Nate Foster, and the POPL reviewers gave us helpful comments. 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

Matthias Blume and David A. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

Olaf Chitil and Frank Huch. Monadic, prompt lazy assertions in haskell. In *APLAS*, pages 38–53, 2007.

Robert Bruce Findler. Contracts as pairs of projections. In *Symposium on Logic Programming*, pages 226–241, 2006.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, 2002.

Cormac Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.

Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, 2007.

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

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS*, pages 29–40, 2007.

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

Kenneth Knowles and Cormac Flanagan. Hybrid type checking. To appear in TOPLAS., 2009.

Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

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

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

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

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *Principles of Programming Languages (POPL)*, pages 41–52, 2009.