

Contracts Made Manifest

Michael Greenberg

University of Pennsylvania

Benjamin C. Pierce

University of Pennsylvania

Stephanie Weirich

University of Pennsylvania

Abstract

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

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

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

1. Introduction

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

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

“wraps” the function $\lambda x:\text{Int}. \text{pred } x$ in a pair of checks: whenever the wrapped function is called, the argument is checked to see whether it is nonzero and, if not, the blame term $\uparrow l'$ is produced, signaling that the *context* of the contracted term violated the expectations of the contract. If the argument check succeeds, then the

function is run and its result is checked against the contract $\text{pos } x$, raising $\uparrow l$ if this fails (e.g., if the wrapped function is applied to 1).

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

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

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

$$f = \langle I \rightarrow I \Rightarrow P \rightarrow P \rangle^l (\lambda x:I. \text{pred } x)$$

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

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

for checking that arguments satisfy S_1 will be discharged elsewhere (with another cast, with a different blame label).

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

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

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

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

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

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

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

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

where $I = \{x:\text{Int} \mid \text{true}\}$ and $N = \{x:\text{Int} \mid \text{nonzero } x\}$. This rather strange contract has the basic shape $f:c_1 \mapsto c_2$, where c_2 uses f in a way that violates c_1 ! In particular, if we apply it to

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

$$\begin{aligned} B &::= \text{Bool} \mid \dots \\ k &::= \text{true} \mid \text{false} \mid \dots \end{aligned}$$

Figure 1. Base types and constants for λ_C and λ_H

Types and contracts

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

Terms, values, results, and evaluation contexts

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

Figure 2. Syntax for λ_C

$\lambda f:\text{Int} \rightarrow \text{Int}. 0$ and then apply the result to $\lambda x:\text{Int}. x$ and 5, the final result will be 5, since $\lambda x:\text{Int}. x$ does satisfy the contract $\{x:\text{Int} \mid \text{nonzero } x\} \mapsto \{y:\text{Int} \mid \text{true}\}$ and 5 satisfies the contract $\{z:\text{Int} \mid (\lambda x:\text{Int}. x) 0 = 0\}$. However, the translation of f into λ_H inserts an extra check, wrapping the occurrence of f in the codomain contract with a cast from $N \rightarrow I$ to $I \rightarrow I$, which fails when the wrapped function is applied to 0. We discuss this in greater detail in Section 6.

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

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

As a warm-up, we begin with the nondependent versions of λ_C and λ_H and (in the next section) the translations between them. The dependent languages, dependent translations, and their properties are developed in Sections 4, 5, and 6.

2.1 The language λ_C

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

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

$\overline{k \ v \longrightarrow_c \llbracket k \rrbracket(v)}$	E_CONST
$\overline{(\lambda x: T_1. t_2) \ v \longrightarrow_c \ t_2\{x := v\}}$	E_BETA
$\overline{\langle \{x: B \mid t\} \rangle^{l, l'} \ k \longrightarrow_c \langle \{x: B \mid t\}, t\{x := k\}, k \rangle^l}$	E_CCHECK
$\overline{\langle \{x: B \mid t\}, \text{true}, k \rangle^l \longrightarrow_c \ k}$	E_OK
$\overline{\langle \{x: B \mid t\}, \text{false}, k \rangle^l \longrightarrow_c \uparrow l}$	E_FAIL
$\overline{\langle \langle c_1 \mapsto c_2 \rangle^{l, l'} \ v \rangle \ v' \longrightarrow_c \langle c_2 \rangle^{l, l'} (v (\langle c_1 \rangle^{l', l} v'))}$	E_CDECOMP
$\frac{t_1 \longrightarrow_c t_2}{E[t_1] \longrightarrow_c E[t_2]}$	E_COMPAT
$\overline{E[\uparrow l] \longrightarrow_c \uparrow l}$	E_BLAZE

Figure 3. Operational semantics for λ_C

Values v comprise abstractions, contracts, function contracts applied to values, and constants; a *result* r is either a value or $\uparrow l$ for some l . We define constants using three constructions: the set \mathcal{K}_B , which contains constants of base type B ; the type-assignment function ty_c , which maps constants to first-order types of the form $B_1 \rightarrow B_2 \rightarrow \dots \rightarrow B_n$ (and which is assumed to agree with \mathcal{K}_B); and the denotation function $\llbracket - \rrbracket$ which maps constants to functions from constants to results (and must agree with ty_c). Denotations map to results to allow for partiality. We assume that Bool is among the base types, with $\mathcal{K}_{\text{Bool}} = \{\text{true}, \text{false}\}$.

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

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

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

The typing rules for λ_C (Figure 4) are straightforward, assigning expressions simple types. We give types to constants using the type-assignment function ty_c . Blame expressions have all types. Contracts are checked for well-formedness using the judgment

$\boxed{\Gamma \vdash t : T}$	$\frac{x: T \in \Gamma}{\Gamma \vdash x : T}$	T_VAR
	$\overline{\Gamma \vdash k : \text{ty}_c(k)}$	T_CONST
	$\frac{\Gamma, x: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x: T_1. t_2 : T_1 \rightarrow T_2}$	T_LAM
	$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 \ t_2 : T_2}$	T_APP
	$\frac{\vdash_c c : T}{\Gamma \vdash \langle c \rangle^{l, l'} : T \rightarrow T}$	T_CONTRACT
	$\overline{\Gamma \vdash \uparrow l : T}$	T_BLAZE
	$\frac{\emptyset \vdash k : B \quad \emptyset \vdash t_2 : \text{Bool} \quad \vdash_c \{x: B \mid t_1\} : B \quad \vdash t_2 \supset t_1\{x := k\}}{\emptyset \vdash \langle \{x: B \mid t_1\}, t_2, k \rangle^l : B}$	T_CHECKING
$\boxed{\vdash_c c : T}$	$\frac{x: B \vdash t : \text{Bool}}{\vdash_c \{x: B \mid t\} : B}$	T_BASEC
	$\frac{\vdash_c c_1 : T_1 \quad \vdash_c c_2 : T_2}{\vdash_c c_1 \mapsto c_2 : T_1 \rightarrow T_2}$	T_FUNC
$\boxed{\vdash t_2 \supset t_1}$	$\frac{t_1 \longrightarrow^* \text{true} \text{ implies } t_2 \longrightarrow^* \text{true}}{\vdash t_1 \supset t_2}$	T_IMP

Figure 4. Typing rules for λ_C

$\vdash_c c : T$, comprising the rules T.BASEC, which requires that the checking term in a base contract return a boolean value when supplied with a term of the right type, and T.FUNC. Note that the predicate t in a contract $\{x: B \mid t\}$ can contain at most x free, since we are talking about non-dependent contracts here. Contract application, like function application, is checked using T.APP.

The T.CHECKING rule only applies in the empty context—all that is needed because active checks are a technical device that should not be used in source programs. The rule ensures that the contract $\{x: B \mid t_1\}$ 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 $\langle \{x: \text{Int} \mid \text{pos } x\}, \text{true}, 0 \rangle^l$ should not be well typed; and second, we use this property in showing that the translations are type preserving (see Theorem 5.2 for ψ and Section 6 for ϕ). This rule obviously makes typechecking for the full “internal language” with checks undecidable, but excluding checks recovers decidability.

2.2 The language λ_H

Our second core calculus, nondependent λ_H , notably includes *refinement types* and *cast expressions*. The syntax appears in Figure 5. Unlike λ_C , which separates contracts from types, λ_H combines them into refined base types $\{x: B \mid s_1\}$ and normal function

Types

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

Terms, values, results, and evaluation contexts

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

Figure 5. Syntax for λ_H

$\frac{}{k w \rightarrow_h \llbracket k \rrbracket(w)}$	F_CONST
$\frac{}{(\lambda x:S_1. s_2) w_2 \rightarrow_h s_2 \{x := w_2\}}$	F_BETA
$\frac{}{\langle \{x:B \mid s_1\} \Rightarrow \{x:B \mid s_2\} \rangle^l k \rightarrow_h \langle \{x:B \mid s_2\}, s_2 \{x := k\}, k \rangle^l}$	F_CCHECK
$\frac{}{\langle \{x:B \mid s\}, \text{true}, k \rangle^l \rightarrow_h k}$	F_OK
$\frac{}{\langle \{x:B \mid s\}, \text{false}, k \rangle^l \rightarrow_h \uparrow l}$	F_FAIL
$\frac{}{\langle \langle S_{11} \rightarrow S_{12} \Rightarrow S_{21} \rightarrow S_{22} \rangle^l w \rangle^l \rightarrow_h \langle S_{12} \Rightarrow S_{22} \rangle^l (w \langle \langle S_{21} \Rightarrow S_{11} \rangle^l w') \rangle^l}$	F_CDECOMP
$\frac{s_1 \rightarrow_h s_2}{F[s_1] \rightarrow_h F[s_2]}$	F_COMPAT
$\frac{}{F[\uparrow l] \rightarrow_h \uparrow l}$	F_BLAZE

Figure 6. Operational semantics for λ_H

types $S_1 \rightarrow S_2$. As in Section 2.1, 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 written with a trivial refinement, as the *raw* type $\{x:B \mid \text{true}\}$. The terms of the language are mostly standard, including variables, the same first-order constants as λ_C , blame, abstractions, and applications. The cast expression $\langle S_1 \Rightarrow S_2 \rangle^l$ dynamically checks that a given term of type S_1 can be given type S_2 . Like λ_C , this language uses active checks to give a small-step semantics to cast expressions. Note that we only use a single label for casts, following the original formulation of hybrid types [Flanagan 2006].

The values of λ_H comprise constants, abstractions, casts, and function casts applied to values. Results q are either values or blame. We give meaning to constants as we did in λ_C , reusing the denotation function $\llbracket - \rrbracket$. Type assignment is via ty_h (which we assume produces well-formed types, defined in Figure 7). To keep the languages in sync, we require that ty_h and ty_c agree on “type skeletons”: if $\text{ty}_c(k) = B_1 \rightarrow B_2$, then $\text{ty}_h(k) = \{x:B_1 \mid s_1\} \rightarrow \{x:B_2 \mid s_2\}$.

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

It is worth observing how the decomposition rules compare. In λ_C , the term $\langle \langle c_1 \mapsto c_2 \rangle^{l,l'} v \rangle^l w'$ decomposes in a straightforward way: c_1 checks the argument v' and c_2 checks the result of the application. In λ_H the term $\langle \langle S_{11} \rightarrow S_{12} \Rightarrow S_{21} \rightarrow S_{22} \rangle^l w \rangle^l w'$ decomposes to two casts. The contravariant check $\langle S_{12} \Rightarrow S_{22} \rangle^l w'$ makes w' a suitable input for w , while $\langle S_{11} \Rightarrow S_{21} \rangle^l w$ checks the result from w applied to (the cast) w' . Suppose $S_{21} = \{x:\text{Int} \mid \text{pos } x\}$ and $S_{11} = \{x:B \mid \text{nonzero } x\}$. Then the check on the argument ensures that $\text{nonzero } x \xrightarrow_h^* \text{true}$ —not, as one might expect, that $\text{pos } w' \xrightarrow_h^* \text{true}$. 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 when a contract fails is more obvious with casts. The translations below will need to reconcile these facts, carefully encoding correct checking and blame behavior.

The typing rules for λ_H (Figure 7) are also similar to those of λ_C . Just as the λ_C rule T_CONTRACT checks to make sure that the contract has the right form, the λ_H rule S_CAST ensures that the two types in a cast expression have the same simple-type skeletons.

$$\begin{aligned} \llbracket \{x:B \mid s\} \rrbracket &= B \\ \llbracket S_1 \rightarrow S_2 \rrbracket &= \llbracket S_1 \rrbracket \rightarrow \llbracket S_2 \rrbracket \end{aligned}$$

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

Type well-formedness is similar to contract well-formedness in λ_C , though the WF_RAW case is added to get us off the ground.

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

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

We do not consider source programs with subtyping—that would make the type system undecidable—but it is necessary for preservation.² Consider the term:

$$\langle \{x:\text{Int} \mid x = 1\} \Rightarrow \{x:\text{Int} \mid \text{pos } x\} \rangle^l 1 \xrightarrow_h^* 1$$

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

Standard progress and preservation theorems also hold for λ_H .

3. The nondependent translations

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

²Trade-offs between static subtype checking and dynamic predicate checking that allow decidable systems with subtyping are discussed much more fully in Flanagan [2006] and Knowles and Flanagan [2009].

$\Delta \vdash s : S$		
	$\frac{x:S \in \Delta}{\Delta \vdash x : S}$	S_VAR
	$\frac{}{\Delta \vdash k : \text{ty}_h(k)}$	S_CONST
	$\frac{\vdash S_1 \quad \Delta, x:S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x:S_1. s_2 : S_1 \rightarrow S_2}$	S_LAM
	$\frac{\Delta \vdash s_1 : S_1 \rightarrow S_2 \quad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1 s_2 : S_2}$	S_APP
	$\frac{\vdash S_1 \quad \vdash S_2 \quad [S_1] = [S_2]}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \rightarrow S_2}$	S_CAST
	$\frac{\vdash S}{\Delta \vdash \uparrow l : S}$	S_BLAZE
	$\frac{\Delta \vdash s : S_1 \quad \vdash S_2 \quad \vdash S_1 <: S_2}{\Delta \vdash s : S_2}$	S_SUB
	$\frac{\emptyset \vdash k : \{x:B \mid \text{true}\} \quad \emptyset \vdash s_2 : \{x:\text{Bool} \mid \text{true}\}}{\emptyset \vdash \langle \{x:B \mid s_1\}, s_2, k \rangle^l : \{x:B \mid s_1\}}$	S_CHECKING
$\vdash S_1 <: S_2$		
	$\frac{\forall k \in \mathcal{K}_B. \vdash s_1 \{x := k\} \supset s_2 \{x := k\}}{\vdash \{x:B \mid s_1\} <: \{x:B \mid s_2\}}$	SUB_BASE
	$\frac{\vdash S_{21} <: S_{11} \quad \vdash S_{12} <: S_{22}}{\vdash S_{11} \rightarrow S_{12} <: S_{21} \rightarrow S_{22}}$	SUB_FUN
$\vdash s_1 \supset s_2$		
	$\frac{s_1 \xrightarrow{*}_h \text{true implies } s_2 \xrightarrow{*}_h \text{true}}{\vdash s_1 \supset s_2}$	S_IMP
$\vdash S$		
	$\frac{}{\vdash \{x:B \mid \text{true}\}}$	WF_RAW
	$\frac{x:\{x:B \mid \text{true}\} \vdash s : \{x:\text{Bool} \mid \text{true}\}}{\vdash \{x:B \mid s\}}$	WF_REFINE
	$\frac{\vdash S_1 \quad \vdash S_2}{\vdash S_1 \rightarrow S_2}$	WF_FUN

Figure 7. Typing rules for λ_H

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

For ψ , translating from λ_H 's rich types to λ_C 's simple types is easy: we just erase the types to their simple skeletons. The interesting case is how we translate the cast $\langle S_1 \Rightarrow S_2 \rangle^l$ to the

contract $\langle \psi(S_1, S_2) \rangle^{l,l'}$ by translating the pair of types together. Note that ψ can either translate a λ_H term to a λ_C term or *two* λ_H types to a λ_C contract.

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

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

$$\begin{aligned} \langle \{x:\text{Int} \mid \text{nonzero } x\} \rightarrow \{y:\text{Int} \mid \text{true}\} \rangle &\Rightarrow \\ \langle \{x:\text{Int} \mid \text{true}\} \rightarrow \{y:\text{Int} \mid \text{pos } y\} \rangle^l & \end{aligned}$$

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

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

$$\begin{aligned} \lceil \{x:B \mid t\} \rceil &= \{x:B \mid \text{true}\} \\ \lceil c_1 \mapsto c_2 \rceil &= \lceil c_1 \rceil \rightarrow \lceil c_2 \rceil \end{aligned}$$

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:

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

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

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

It translates to this λ_H term:

$$\begin{aligned} \lambda f:\lceil \text{Int} \rightarrow \text{Int} \rceil. \\ \langle \{x:\text{Int} \mid \text{nonzero } x\} \rightarrow \{y:\text{Int} \mid \text{pos } y\} \rangle^{l'} &\Rightarrow \lceil \text{Int} \rightarrow \text{Int} \rceil^{l'} \\ (\langle \lceil \text{Int} \rightarrow \text{Int} \rceil \Rightarrow \{x:\text{Int} \mid \text{nonzero } x\} \rightarrow \{y:\text{Int} \mid \text{pos } y\} \rangle^l f) & \end{aligned}$$

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

Contracts and contexts

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

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

Operational Semantics

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

Typing rules

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

$$\frac{x:c \in \Gamma}{\Gamma \vdash x : [c]} \text{T_VARC}$$

$$\frac{\Gamma \vdash_c c_1 : T_1 \quad \Gamma, x:c_1 \vdash_c c_2 : T_2}{\Gamma \vdash_c x:c_1 \mapsto c_2 : T_1 \rightarrow T_2} \text{T_FUNC}$$

Figure 8. Changes for dependent λ_C

example, we translate $\langle\langle x:\text{Int} \mid \text{nonzero } x \rangle\rangle^{l,l'}$ to this:

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

$$\langle\langle x:\text{Int} \mid \text{nonzero } x \rangle\rangle \Rightarrow [\text{Int}]^{l'}$$

$$\langle\langle [\text{Int}] \Rightarrow \{x:\text{Int} \mid \text{nonzero } x \} \rangle\rangle^{l} x$$

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

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

4. The dependent languages

We now extend λ_C to dependent function contracts and λ_H to dependent functions. The changes are summarized in Figures 8 and 9. Very little needs to be changed in λ_C , since contracts and types barely interact; the changes to E_CDECOMP and T_FUNC are the important ones. Adding dependency to λ_H is more involved. In particular, adding contexts to the subtyping judgment entails adding contexts to S_IMP. To avoid a dangerous circularity, we must define closing substitutions in terms of a separate type semantics. Additionally, the new F_CDECOMP rule has a somewhat unintuitive (but necessary) asymmetry, as we explain in Section 4.2.

4.1 Dependent λ_C

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

The only changes to the system of Section 2.1 to add dependency to λ_C are the new T_FUNC and E_CDECOMP rules. We also make some small changes to the bindings in contexts: T_FUNC adds $x:c_1$ to the context when checking the codomain of a function contract, and we split the variable rule in two. Both of these changes help ϕ preserve types. (See Section 6).

Types

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

Operational semantics

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

Typing rules

$$\frac{\Delta \vdash s_1 : (x:S_1 \rightarrow S_2) \quad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1 s_2 : S_2 \{x := s_2\}} \text{S_APP}$$

$$\frac{\Delta, x:\{x:B \mid \text{true}\} \vdash s : \{x:\text{Bool} \mid \text{true}\}}{\Delta \vdash \{x:B \mid s\}} \text{WF_REFINE}$$

$$\frac{\Delta \vdash S_1 \quad \Delta, x:S_1 \vdash S_2}{\Delta \vdash x:S_1 \rightarrow S_2} \text{WF_FUN}$$

$$\frac{\Delta \vdash S_{21} <: S_{11} \quad \Delta, x:S_{21} \vdash S_{12} <: S_{22}}{\Delta \vdash x:S_{11} \rightarrow S_{12} <: x:S_{21} \rightarrow S_{22}} \text{SUB_FUN}$$

$$\frac{\forall \sigma. ((\Delta \models \sigma \text{ and } \sigma(s_1) \longrightarrow_h^* \text{true}) \text{ implies } \sigma(s_2) \longrightarrow_h^* \text{true})}{\Delta \vdash s_1 \Rightarrow s_2} \text{S_IMP}$$

Closing Substitutions

$$\frac{}{\emptyset \models \emptyset} \text{CS_EMPTY}$$

$$\frac{s \in \llbracket S \rrbracket \quad \Delta \{x := s\} \models \sigma}{x:S, \Delta \models \sigma \{x := s\}} \text{CS_EXT}$$

Figure 9. Changes for dependent λ_H

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

4.2 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-

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 preservation and progress theorems follow directly from the results for Knowles and Flanagan's language, since CBV is a restriction of full β -reduction. However, note that the reduction relation is used in the type system (in rule S_IMP), so the type systems for the two languages are not the same. For example, suppose that the term *bad* contains a cast that fails. In our system the type $\{y:B \mid \text{true}\}$ is not a subtype of $\{y:B \mid (\lambda x:S. \text{true}) \text{ bad}\}$ because the contract evaluates to

Denotations of types and kinds

$$\begin{aligned}
s \in \llbracket \{x:B \mid s_0\} \rrbracket &\iff s \xrightarrow*_h \uparrow l \\
&\quad \vee (\exists k \in \mathcal{K}_B. s \xrightarrow*_h k \\
&\quad \quad \wedge s_0\{x := k\} \xrightarrow*_h \text{true}) \\
s \in \llbracket x:S_1 \rightarrow S_2 \rrbracket &\iff \forall q \in \llbracket S_1 \rrbracket. s q \in \llbracket S_2\{x := q\} \rrbracket \\
\{x:B \mid s\} \in \llbracket \star \rrbracket &\iff \forall k \in \mathcal{K}_B. \\
&\quad s\{x := k\} \in \llbracket \{x:\text{Bool} \mid \text{true}\} \rrbracket \\
x:S_1 \rightarrow S_2 \in \llbracket \star \rrbracket &\iff S_1 \in \llbracket \star \rrbracket \\
&\quad \wedge \forall q \in \llbracket S_1 \rrbracket. S_2\{x := q\} \in \llbracket \star \rrbracket
\end{aligned}$$

Semantic judgments

$$\begin{aligned}
\Delta \models S_1 <: S_2 &\iff \forall \sigma \text{ s.t. } \Delta \models \sigma : \\
&\quad \llbracket \sigma(S_1) \rrbracket \subseteq \llbracket \sigma(S_2) \rrbracket \\
\Delta \models s : S &\iff \sigma(s) \in \llbracket \sigma(S) \rrbracket \\
\Delta \models S &\iff \sigma(S) \in \llbracket \star \rrbracket
\end{aligned}$$

Figure 10. Type and kind semantics for dependent λ_H

by-value evaluation. However, the benefit of a CBV semantics is a better treatment of blame. By contrast, Knowles and Flanagan [2009] cannot treat failed casts as exceptions because that would destroy confluence. The needed extensions are detailed in Figures 9 and 10.⁴

First, we enrich the type system with dependent function types, $x:S_1 \rightarrow S_2$, where x may appear in S_2 . This dependency must be maintained through higher-order casts in the rule F_CDECOMP. As the cast decomposes, the variables in the codomain types of such a cast must be replaced by the argument. However, this substitution is asymmetric; on one side, we cast that argument and on the other we do not. This behavior is required for type preservation. For suppose we have $\Delta \vdash x:S_{11} \rightarrow S_{12}$ and $\Delta \vdash x:S_{21} \rightarrow S_{22}$ with equal skeletons, and values $\Delta \vdash w : (x:S_{11} \rightarrow S_{12})$ and $\Delta \vdash w' : S_{21}$. Then $\Delta \vdash (\langle x:S_{11} \rightarrow S_{12} \Rightarrow x:S_{21} \rightarrow S_{22} \rangle^l w) w' : S_{22}\{x := w'\}$. To preserve variable scoping when we decompose the cast, we must make *some* substitution into S_{12} and S_{22} , but which? It is clear that we must substitute w' into S_{22} , since the original application has type $S_{22}\{x := w'\}$. Decomposing the cast will produce the inner application $\Delta \vdash w (\langle S_{21} \Rightarrow S_{11} \rangle^l w') : S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l w'\}$. Thus, to be able to apply the codomain cast, we must substitute $\langle S_{21} \Rightarrow S_{11} \rangle^l w'$ into S_{12} . This calculation tells us exactly how F_CDECOMP has to look.

Next, we change the typing rules to admit dependency. The new application rule, S_APP, substitutes the argument for the variable in the result of an application. We generalize the refinement-type formation rule, WF_REFINE, to allow predicates that use variables from the enclosing context. The formation rule for function types adds the bound variable to the context when checking the codomain. In SUB_FUN, subtyping for dependent function types remains contravariant, but we also add the argument variable to the context with the smaller type.

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

Some care is needed here to prevent the typing rules from becoming circular: the typing rule S_SUB references the subtyping judgment. The subtyping rule SUB_REFINE references the implica-

blame. However, the subtyping does hold in the Knowles and Flanagan system because in that language the predicate reduces to true.

⁴The semantics in these figures is the same as that of Flanagan [2006] except for the evaluation relation, the treatment of blame as an exception, and a change to the type semantics that we discuss below.

tion judgment. The single implication rule S_IMP has $\Delta \models \sigma$ in a negative position. To guarantee the well-definedness of the type system, we must not allow $\Delta \models \sigma$ to refer back to the other judgments.

To avoid circularity, we define a *denotational semantics* for λ_H 's types.⁵ The basic idea is that the semantics of a type is a set of closed terms that is defined independently of the syntactic typing relation, but that turns out to contain all closed well-typed terms of that type. Thus, in the definition of $\Delta \models \sigma$, we quantify over a somewhat larger set than necessary—not just the syntactically well-typed terms of appropriate type (which are all the ones that will ever appear in programs), but all semantically well-typed ones.

The type semantics appears in Figure 10. It is defined by induction on the structure of type skeletons. For refinement types, terms must either go to blame or produce a constant that satisfies (all instances of) the given predicate. For function types, well-typed arguments must go to well-typed results. (Notice that, by construction, these sets include only terminating terms that do not get stuck.)

4.1 Lemma [Strong normalization]: If $s \in \llbracket S \rrbracket$, then there exists a q such that $s \xrightarrow*_h q$, i.e., either $s \xrightarrow*_h w$ or $s \xrightarrow*_h \uparrow l$.

What we want to know about the type semantics is *semantic type soundness*: if $\emptyset \vdash s : S$, then $s \in \llbracket S \rrbracket$. However, to prove this, we must generalize it. In the bottom of Figure 10, we define three *semantic judgements* that correspond to each of the three typing judgments. (Note that the third one requires the definition of a *kind* semantics that picks out well-behaved types—those whose embedded contracts belong to the type semantics.) We then show that the typing judgments imply their semantic counterparts.

4.2 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 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 $s_1 \in \llbracket x:S_1 \rightarrow S_2 \rrbracket$ and $s_2 \in \llbracket S_1 \rrbracket$. According to S_APP, the application's type is $S_2\{x := s_2\}$. By the type semantics defined in Figure 10, if $s_1 \in \llbracket x:S_1 \rightarrow S_2 \rrbracket$, then $s_1 q \in \llbracket S_2\{x := q\} \rrbracket$ for any $q \in \llbracket S_1 \rrbracket$. Sadly, s_2 isn't necessarily a result! We do know, however, that $s_2 \in \llbracket S_1 \rrbracket$, so $s_2 \xrightarrow*_h q_2$ 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 \Rightarrow 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. The definition of parallel reduction is standard except that we need to be careful to make it respect our call-by-value reduction order: the *beta*-redex $(\lambda x:S_1. s_1) s_2$ should not be contracted (though other redices within s_1 and s_2 can be) unless s_2 is a value, since this can change

⁵Knowles 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.

Term translation

$$\begin{aligned}
\psi(x) &= x \\
\psi(k) &= k \\
\psi(\lambda x:S. s) &= \lambda x:[S]. \psi(s) \\
\psi(s_1 s_2) &= \psi(s_1) \psi(s_2) \\
\psi(\uparrow l) &= \uparrow l \\
\psi(\langle S_1 \Rightarrow S_2 \rangle^l) &= \langle \psi^l(S_1, S_2) \rangle^{l,l} \\
\psi(\langle \{x:B \mid s_1\}, s_2, k \rangle^l) &= \langle \{x:B \mid \psi(s_1)\}, \psi(s_2), k \rangle^l
\end{aligned}$$

Type-to-contract translation

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

Figure 11. ψ mapping dependent λ_H to dependent λ_C

the order of effects. The proof requires a longish sequence of technical lemmas that essentially show that \Rightarrow commutes with \longrightarrow_h^* . Details can be found in the long version.

4.3 Theorem: [Parallel reduction preserves type semantics] If $S_1 \Rightarrow^* S_2$ then $\llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket$.

Once the semantic proof of type soundness is done, we can lift it to a standard proof of progress and preservation, using semantic soundness to handle the `S_IMP` case of the substitution lemma. However, the argument is still not completely straightforward: a similar problem arises in the proof of preservation, again in the application case. Consider a well-typed application $\Delta \vdash s_1 s_2 : S_2\{x := s_2\}$. Suppose $s_2 \longrightarrow_h s_2'$. It is easy to reconstruct $\Delta \vdash s_1 s_2 : S_2\{x := s_2'\}$, but that is not what we must show for preservation. How do $S_2\{x := s_2\}$ and $S_2\{x := s_2'\}$ relate? The semantic result of Lemma 4.3 is not enough—preservation requires us to be able to generate the typing derivation in the *syntactic* type system. To this end, we establish another property of parallel reduction: types that parallel-reduce are subtypes both ways round.

4.4 Lemma: If $\Delta \vdash S_1$ and $S_1 \Rightarrow S_2$ then $\Delta \vdash S_1 <: S_2$ and $\Delta \vdash S_2 <: S_1$.

We can then use `S_SUB` to show $\Delta \vdash s_1 s_2' : S_2\{x := s_2'\}$, giving us the hard case of the preservation proof.

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

The proof of progress, fortunately, is entirely straightforward.

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

5. Translating λ_H to λ_C : dependent ψ

In this section, we formally define ψ for the dependent versions of 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 11. Most terms are translated homomorphically: variables, constants, applications, and active checks are all translated straightforwardly. We translate abstractions compatibly, translating the annotation by erasing the refined λ_H type to its simple-type skeleton. As mentioned in Section 3, the trickiest part is the translation of casts between function types: we are careful to mimic the behavior of the `F_CDECOMP` rule—when generating the codomain contract from a cast between two function types, we perform the same asymmetric substitution

Result correspondence

$$\begin{aligned}
r \approx q : B &\iff \\
r = q = \uparrow l &\vee \\
r = q = k \in \mathcal{K}_B & \\
r \approx q : T_1 \rightarrow T_2 &\iff \\
r = q = \uparrow l &\vee \\
(r = v \wedge q = v &\wedge \\
\forall t' \sim s' : T_1. r t' \sim q s' : T_2) &
\end{aligned}$$

Term correspondence

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

Contract / type correspondence

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

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

Figure 12. Blame-exact correspondence

as `F_CDECOMP`. Since ψ on casts inserts new casts, we need to pick a blame label to use: $\psi(\langle S_1 \Rightarrow S_2 \rangle^l)$ passes l as an index to $\psi^l(S_1, S_2)$ to use when generating casts.

We prove that s and $\psi(s)$ behaviorally correspond using logical relations. We then show that ψ is type preserving—though we have to use the logical relation to do so, since the typing rules `T_CHECKING` and `S_CHECKING` have operational content.

We define the *term correspondence* relation $t \sim s : T$ in Figure 12; it is a logical relation, defined as a fixpoint over its λ_C -type index. The *contract/cast correspondence* $c \sim S_1 \Rightarrow^l S_2 : T$ relates contracts and pairs of λ_H types. We define it using the term correspondence in the base type case, and following the pattern of `F_CDECOMP` in the function case. Since it inserts a cast in the function cast, we index it with a label, just like ψ . Note that the correspondence is blame-exact, relating λ_C and λ_H terms that either blame the same label or go to corresponding values. We define closing substitutions ignoring the contracts in the context; we lift the relation to open terms in the standard way. The behavioral correspondence is quite strong: it is easy to check that, if $t \sim s : B$, then t and s either both go to $k \in \mathcal{K}_B$ or both go to $\uparrow l$.

We can use the correspondence relations to show that s and its translation $\psi(s)$ correspond behaviorally—i.e., that ψ faithfully translates the λ_H semantics.

5.1 Theorem [Behavioral correspondence]:

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

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

- ### 5.2 Theorem [Type preservation for ψ]:
1. If $\psi(s) = t$ and $\Delta \vdash s : S$ then $\llbracket \Delta \rrbracket \vdash t : \llbracket S \rrbracket$
 2. If $\psi^l(S_1, S_2) = c$ and $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $\llbracket S_1 \rrbracket = \llbracket S_2 \rrbracket = T$, then $\llbracket \Delta \rrbracket \vdash_c c : T$.

Terms		Contexts	
$\phi(\Gamma_1, x:T, \Gamma_2 \vdash x : T)$	$= x$	$\phi(\vdash \emptyset)$	$= \emptyset$
$\phi(\Gamma_1, x:c, \Gamma_2 \vdash x : [c])$	$= \langle \phi(\Gamma_1 \vdash_c c : [c]) \Rightarrow [c] \rangle^{l_0} x$	$\phi(\vdash \Gamma, x:T)$	$= \phi(\vdash \Gamma), x:[T]$
$\phi(\Gamma \vdash k : T)$	$= k$	$\phi(\vdash \Gamma, x:c)$	$= \phi(\vdash \Gamma), x:\phi(\Gamma \vdash_c c : [c])$
$\phi(\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2)$	$= \lambda x:[T_1]. \phi(\Gamma, x:T_1 \vdash t_2 : T_2)$		
$\phi(\Gamma \vdash t_1 t_2 : T_2)$	$= \phi(\Gamma \vdash t_1 : T_1 \rightarrow T_2) \phi(\Gamma \vdash t_2 : T_1)$		
$\phi(\Gamma \vdash \uparrow l : T)$	$= \uparrow l$		
$\phi(\emptyset \vdash \langle c, t, k \rangle^l : B)$	$= \langle \phi(\emptyset \vdash_c c : B), \phi(\emptyset \vdash t : B), k \rangle^l$		
$\phi(\Gamma \vdash \langle c \rangle^{l,l'} : T)$	$= \lambda x:[c]. \langle \phi(\Gamma \vdash_c c : T) \Rightarrow [c] \rangle^{l'} (\langle [c] \Rightarrow \phi(\Gamma \vdash_c c : T) \rangle^l x)$		where x is fresh
Types			
$\phi(\Gamma \vdash_c \{x:B \mid t\} : B)$	$= \{x:B \mid \phi(\Gamma, x:B \vdash t : \text{Bool})\}$		
$\phi(\Gamma \vdash_c x:c_1 \mapsto c_2 : T_1 \rightarrow T_2)$	$= x:\phi(\Gamma \vdash_c c_1 : T_1) \rightarrow \phi(\Gamma, x:c_1 \vdash_c c_2 : T_2)$		

Figure 13. The translation ϕ from dependent λ_C to dependent λ_H

6. Translating λ_C to λ_H : dependent ϕ

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

The full definition is in Figure 13. One point to note is that, in the dependent case, we need to translate *derivations* of well-formedness and well-typing of λ_C contexts, terms, and contracts into λ_H contexts, terms, and types. We need to use derivations to ensure type preservation. Notice that T.VART and T.VARC derivations are translated differently: variables bound to simple types translate to themselves, but we add a cast (with distinguished label l_0) to variables bound to contracts.

To see why we make this distinction, consider the function contract $f:(x:\{x:\text{Int} \mid \text{pos } x\} \mapsto \{y:\text{Int} \mid \text{true}\}) \mapsto \{z:\text{Int} \mid f 0 = 0\}$. Note that this contract is well-formed in λ_C , but that the codomain “abuses” the bound variable. A naïve translation will *not* be well-typed in λ_H , since $f 0$ will not be typeable in the context $f:(x:\{x:\text{Int} \mid \text{pos } x\} \rightarrow \{y:\text{Int} \mid \text{true}\})$, $z:[\text{Int}]$ — f only accepts positive arguments. The problem is that WF.FUN can add a non-raw type to the context, so we need to restore the “variables have raw types” invariant. By tracking which variables were bound by contracts in λ_C , we can be sure to cast them to raw types when they’re referenced—this motivates the $x:c$ binding form in dependent λ_C . We therefore translate the contract above to $f:S \rightarrow \{z:\text{Int} \mid (\langle S \Rightarrow [\text{Int} \rightarrow \text{Int}] \rangle^{l_0} f) 0 = 0\}$, where $S = x:\{x:\text{Int} \mid \text{pos } x\} \rightarrow \{y:\text{Int} \mid \text{true}\}$.

Constants translate to themselves. One technical point is that, to maintain the raw type invariant, we need λ_H ’s higher-order constants to have typings that can be seen as raw by the subtyping relation, i.e., $\Delta \vdash \text{ty}_h(k) <: [\text{ty}_c(k)]$. This slightly restricts the types we might assign to our constants, e.g., we cannot say $\text{ty}_h(\text{sqrt}) = x:\{x:\text{Float} \mid x = 0\} \rightarrow \{y:\text{Float} \mid (y * y) = x\}$, since it is not the case that $\Delta \vdash \text{ty}_h(\text{sqrt}) <: [\text{Float} \rightarrow \text{Float}]$. Since its domain cannot be refined, $[\text{sqrt}]$ must be defined for all $k \in \mathcal{K}_{\text{Float}}$, e.g., $[\text{sqrt}](-1)$ 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 $\llbracket - \rrbracket$, so the subtyping requirement does not seem too severe. We could instead translate k to $\langle \text{ty}_h(k) \Rightarrow [\text{ty}_h(k)] \rangle^{l_0} k$; however, in this case the nondependent fragments of the languages would no longer correspond exactly.

We show the behavioral correspondence using a blame-inexact logical relation, defined in Figure 14. We read $\Gamma \vdash t \sim_{\gamma} s : T$

as “ Γ shows t corresponds to s at T ”. The behavioral correspondence here, though weaker than we had before, is still pretty strong: if $t \sim_{\gamma} s : B$, then either $s \rightarrow_n^* \uparrow l$ or both t and s go to $k \in \mathcal{K}_B$. As in Section 5, we use the term correspondence to define a correspondence relating contracts and λ_H types, and then we lift both correspondences to open terms with dual closing substitutions, also defined in Figure 14. Closing substitutions map variables to terms corresponding at appropriate type. Note that closing substitutions treat the $x:c$ bindings in the context Γ as if they were $x:T$.

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

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

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

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

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

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

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

We can prove type preservation, as well, for terms not containing active checks. We don’t know that translated active checks are well typed because Theorem 6.3 isn’t strong enough to show that $\vdash t_2 \Rightarrow t_1 \{x := k\}$ implies $\emptyset \vdash \phi(\emptyset \vdash t_2 : \text{Bool}) \Rightarrow \phi(x:B \vdash t_1 : \text{Bool}) \{x := k\}$. However, since we only expect these checks to occur at runtime, this is good enough: ϕ preserves the types of source programs.

Result correspondence

$$\begin{aligned} r \approx_{\succ} q : B &\iff \\ r = q = k \in \mathcal{K}_B \vee q = \uparrow l & \\ \\ r \approx_{\succ} q : T_1 \rightarrow T_2 &\iff \\ q = \uparrow l \vee & \\ (r = v \wedge q = w \wedge & \\ \forall t \sim_{\succ} s : T_1. v t \sim_{\succ} w s : T_2) & \end{aligned}$$

Term correspondence

$$t \sim_{\succ} s : T \iff t \xrightarrow{c}^* r \wedge s \xrightarrow{h}^* q \wedge r \approx_{\succ} q : T$$

Contract / type correspondence

$$\begin{aligned} \{x:B \mid t\} \sim_{\succ} \{x:B \mid s\} : B &\iff \\ \forall k \in \mathcal{K}_B. t\{x := k\} \sim_{\succ} s\{x := k\} : \text{Bool} & \\ \\ x:c_1 \mapsto c_2 \sim_{\succ} x:S_1 \rightarrow S_2 : T_1 \rightarrow T_2 &\iff \\ c_1 \sim_{\succ} S_1 : T_1 \wedge & \\ \forall t \sim_{\succ} s : T_1. c_2\{x := t\} \sim_{\succ} S_2\{x := s\} : T_2 & \end{aligned}$$

Dual closing substitutions

$$\Gamma \models_{\succ} \delta \iff \forall x \in \text{dom}(\Gamma). \delta_1(x) \sim_{\succ} \delta_2(x) : [\Gamma(x)]$$

Figure 14. Blame-inexact correspondence

6.4 Theorem [Type preservation]: For programs without active checks:

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

To see that the ϕ we define in Figure 13 does not give us exact blame, let us look at two counterexamples; in both cases, a λ_C term goes to a value while its translation goes to blame. In the first counterexample, blame is raised in λ_H due to F_CDECOMP. In the second, blame is raised due to the extra cast from the translation of T_VARC.

First, let

$$\begin{aligned} c &= f:(x:\{x:\text{Int} \mid \text{true}\} \mapsto \{y:\text{Int} \mid \text{nonzero } y\}) \mapsto \\ &\quad \{z:\text{Int} \mid f 0 = 0\} \\ S_1 &= x:\{x:\text{Int} \mid \text{true}\} \rightarrow \{y:\text{Int} \mid \text{nonzero } y\} \\ S &= \phi(\emptyset \vdash_c c : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}) \\ &= f:S_1 \rightarrow \{z:\text{Int} \mid (\langle S_1 \rangle^{\text{lo}} f) 0 = 0\}. \end{aligned}$$

We find $\langle c \rangle^{\text{li}}$ $(\lambda f.0) (\lambda x.0) \xrightarrow{c}^* 0$ but $(\lambda x: [c]. \langle S \rangle \Rightarrow [S])^{\text{li}} (\langle [S] \Rightarrow S \rangle^{\text{li}} x) (\lambda f.0) (\lambda x.0) \xrightarrow{h}^* \uparrow l$. This is due to the mismatch between E_CDECOMP and F_CDECOMP.

Second, let

$$\begin{aligned} c' &= f:(x:\{x:\text{Int} \mid \text{nonzero } x\} \mapsto \{y:\text{Int} \mid \text{true}\}) \mapsto \\ &\quad \{z:\text{Int} \mid f 0 = 0\} \\ S'_1 &= x:\{x:\text{Int} \mid \text{nonzero } x\} \rightarrow \{y:\text{Int} \mid \text{true}\} \\ S' &= \phi(\emptyset \vdash_c c' : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}) \\ &= f:S'_1 \rightarrow \{z:\text{Int} \mid (\langle S'_1 \rangle^{\text{lo}} f) 0 = 0\}. \end{aligned}$$

We find $\langle c' \rangle^{\text{li}}$ $(\lambda f.0) (\lambda x.0) \xrightarrow{c}^* 0$ but $(\lambda x: [c']. \langle S' \rangle \Rightarrow [c']^{\text{li}}) (\langle [S'] \Rightarrow [c']^{\text{li}} \rangle^{\text{li}} x) (\lambda f.0) (\lambda x.0) \xrightarrow{h}^* \uparrow l_0$. This is due to the extra cast inserted for the sake of type preservation.

The nondependent restriction of ϕ

While ϕ doesn't induce an exact behavioral correspondence in the dependent case, it *does* for nondependent λ_C and λ_H , as Gronski and Flanagan [2007] have also showed. The discrepancy with regard to blame in the full system comes from two sources: the asym-

metric F_CDECOMP rule inserts a cast that E_CDECOMP doesn't, and the translation of T_VARC derivations inserts extra casts. Neither of these come up in the nondependent case: F_CDECOMP does *no* substitution when there's no dependency, and $x:c$ bindings only occur when we have dependent function contracts.

We give a new proof of this fact, different from Gronski and Flanagan's. It follows the same pattern as the dependent case, though of course without the extra cast lemma (6.1). The most significant change is to the correspondence relation—we use the blame-exact correspondence \sim from Figure 12 in Section 5. Since we are able to prove exact behavioral correspondence, we can also show exact type preservation for the full language including active checks. Full details are given in the long version.

7. Related work

Programming languages conferences in recent years have seen a profusion of papers on higher-order contracts and related features. This is all to the good, but, for newcomers to the area, it can be a bit overwhelming—especially given the great variety of technical approaches. To help reduce the level of confusion, in Figure 15 we summarize the important points of comparison between a number of systems that are closely related to ours.

The largest difference (though, oddly, it is rarely discussed) is between latent and manifest treatments of contracts—whether contract checking (or whatever it is called in a given system) is a completely dynamic matter or whether it leaves a “trace” that the type system can track.

Another major distinction (labeled “dep” in the figure) is the presence of dependent contracts (or dependent function types, in manifest systems). Latent systems with dependent contracts also vary in whether their semantics is lax or picky (see Section 4.1).

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

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

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

$$\frac{s_2\{x := k\} \xrightarrow{h}^* \text{true}}{\langle \{x:B \mid s_1\} \Rightarrow \{x:B \mid s_2\} \rangle^{\text{li}} k \xrightarrow{h} k}$$

Such rules are formally awkward, and in any case they violate the spirit of a small-step semantics. Also, the formal definitions of λ_H in both Gronski and Flanagan [2007] and Flanagan [2006] involve a circularity between the typing, subtyping, and implication relations. Knowles and Flanagan [2009] improves the technical presentation of λ_H in both respects. In particular, it introduces (as we do) a denotational interpretation of contract types, avoiding the definitional circularity, and it introduces a new syntactic form of “partially evaluated casts” (like most of the other systems) to maintain a small-step evaluation regime.

Our main contributions are (1) the translations ϕ and ψ and their properties, and (2) the formulation and metatheory of dependent λ_H . (Dependent λ_C is not a contribution on its own: many similar systems have been studied, and in any case its properties are much easier.) The non-dependent part of our ϕ translation essentially coincides with the one studied by Gronski and Flanagan [2007] for

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

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

Figure 15. Comparison of related systems

non-dependent λ_C and λ_H , and our behavioral correspondence theorem is essentially the same as theirs. Our ψ translation completes their story for the non-dependent case, establishing a tight connection between the systems. The full dependent forms of ϕ and ψ are novel, as is the observation that the correspondence between the latent and manifest world is more complex in this setting.

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

The system studied by Ou et al. [2004] is also quite close in spirit to our λ_H . The main difference is that, because their system includes general recursion, they restrict the terms that can appear in contracts to just applications involving predefined constants: only “pure” terms can be substituted into types, and these do not include lambda-abstractions. Our system (like all of the others in Figure 15—see the row labeled “any con”) allows arbitrary user-defined boolean functions to be used as contracts.

Our description of λ_C is ultimately based on λ_{CON} [Findler and Felleisen 2002], though our presentation is slightly different in its use of checks. Hinze et al. [2006] adapted Findler and Felleisen-style to a location-passing implementation in Haskell. Notably, their dependent function contract rule is picky, not lax like ours (and Findler and Felleisen’s).

Our λ_H type semantics in Section 4.2 is effectively a semantics of contracts. Blume and McAllester [2006] offers a semantics of contracts that is slightly different — our semantics includes blame at every type, while theirs explicitly excludes it. Xu et al. [2009] is also similar, though their “contracts” have no dynamic semantics at all—they are simply specifications.

We have discussed only a small sample of the many recent (and classic) papers on contracts and related ideas. We refer the reader

to Knowles and Flanagan [2009] for a more comprehensive survey. Another useful resource is Wadler and Findler [2007] (technically superseded by Wadler and Findler [2009], but with a longer related work section), which surveys work combining contracts with type *Dynamic* and related features.

There are also *many* other systems that use precise types of various sorts in a completely static manner. One notable example is the work of Xu et al. [2009], which uses user-defined boolean predicates to classify values (justifying their use of the term ‘contracts’), but which checks statically that these predicates hold.

It is worth mentioning that Sage [Gronski et al. 2006] and Knowles and Flanagan [2009] both support mixed static and dynamic checking of contracts, using, e.g., a theorem prover. Our work does not address that aspect, since we work with the core calculus λ_H .

8. Future work

Our λ_C and λ_H are strongly normalizing; extending our results to systems that allow recursion is a natural next step. The changes seem nontrivial: with the introduction of nontermination, \sim_{\sim} has to allow not only more blame, but more nontermination in λ_H —running more casts means more opportunities for divergence.

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

Our operational semantics for dependent λ_C used the lax $E_CDECOMP$ rule. Although the picky $E_CDECOMP$ rule is also sound, we have so far been unable to come up with corresponding versions of ψ and ϕ . It seems likely that it is not possible to find translations that establish an exact behavioral correspondence (as in the non-dependent case), but one might hope for a weaker correspondence, analogous to the one we’ve presented here.

Acknowledgments

Sewell and Zappa Nardelli’s OTT tool was invaluable in keeping our technicalities organized. Nate Foster joined us in many early

discussions about contracts. Chris Casinghino and Nate Foster gave us helpful comments on an early draft.

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 Löf. 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.