Polymorphic Contracts

João Filipe Belo[*], Michael Greenberg[*],
Atsushi Igarashi[†], and Benjamin C. Pierce[*]

[*]University of Pennsylvania [†]Kyoto University

Abstract. *Manifest contracts* track precise properties by refining types with predicates—e.g., $\{x{:}\mathsf{Int} \mid x > 0\}$ denotes the positive integers. Contracts and polymorphism make a natural combination: programmers can give strong contracts to abstract types, precisely stating pre- and post-conditions while hiding implementation details—for example, an abstract type of stacks might specify that the pop operation has input type $\{x{:}\alpha\ \mathsf{Stack} \mid \mathsf{not}\,(\mathsf{empty}\,x)\}$. We formalize this combination by defining $\mathrm{F_H}$, a polymorphic calculus with manifest contracts, and establishing fundamental properties including type soundness and relational parametricity. Our development relies on a significant technical improvement over earlier presentations of contracts: instead of introducing a denotational model to break a problematic circularity between typing, subtyping, and evaluation, we develop the metatheory of contracts in a completely syntactic fashion, omitting subtyping from the core system and recovering it *post facto* as a derived property.

Keywords: contracts, refinement types, preconditions, postconditions, dynamic checking, parametric polymorphism, abstract datatypes, syntactic proof, logical relations, subtyping

1 Introduction

Software contracts allow programmers to state precise properties—e.g., that a function takes a non-empty list to a positive integer—as concrete predicates written in the same language as the rest of the program; these predicates can be checked dynamically as the program executes or, more ambitiously, verified statically with the assistance of a theorem prover. Findler and Felleisen [5] introduced "higher-order contracts" for functional languages; these can take one of two forms: predicate contracts like $\{x{:}\mathsf{Int} \mid x > 0\}$, which denotes the positive numbers, and function contracts like $x{:}\mathsf{Int} \rightarrow \{y{:}\mathsf{Int} \mid y \geq x\}$, which denotes functions over the integers that return numbers larger than their inputs.

Greenberg, Pierce, and Weirich [7] contrast two different approaches to contracts: in the *manifest* approach, contracts are types—the type system itself makes contracts 'manifest'; in the *latent* approach, contracts and types live in different worlds (indeed, there may be no types at all, as in PLT Racket's contract system [1]). These two presentations lead to different ways of checking contracts. Latent systems run contracts with checks: for example, $\langle\{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\ n$ checks that $n > 0$. If the check succeeds, then the entire expression will just return n. If it fails, then the entire program will "blame" the label l, raising an uncatchable exception $\Uparrow l$, pronounced "blame l". Manifest systems use casts, $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l$ to convert values from one type to another (the left-hand side is the *source* type and the right-hand side is the *target* type). For predicate contracts, a cast will behave just like a check on the target type: applied to n, the cast either returns n or raises $\Uparrow l$. Checks and casts differ when it comes to function contracts. A function check $(\langle T_1 \to T_2\rangle^l\ v)\ v'$ will reduce to $\langle T_2\rangle^l\ (v\ (\langle T_1\rangle^l\ v'))$, giving v the argument checked at the domain contract and checking that the result satisfies the codomain contract. A function cast $(\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22}\rangle^l\, v)\, v'$ will reduce to $\langle T_{12} \Rightarrow T_{22}\rangle^l\, (v\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, v'))$, wrapping the argument v' in a (contravariant) cast between the domain types and wrapping the result of the application in a (covariant) cast between the codomain types. The differences between checks and casts are discussed at length in [7]. Both presentations have their pros and cons: latent contracts are simpler to design and extend, while manifest contracts make a clearer connection between the static constraints captured by types and the dynamic checks performed by casts. In this work, we consider the manifest approach and endeavor to tame its principal drawback: the complexity of its metatheory. We summarize the issues here, comparing our work to previous approaches more thoroughly in Section 6.

Subtyping is the main source of complexity in the most expressive manifest calculi—those which have dependent functions and allow arbitrary terms in refinements [7, 10]. These calculi have subtyping for two reasons. First, subtyping helps preserve types when evaluating casts with predicate contracts: if $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\ n \longrightarrow^* n$, then we need to type n at $\{x{:}\mathsf{Int} \mid x > 0\}$. Subtyping gives it to us, allowing n to be typed at any predicate contract it satisfies. Second, subtyping can show the equivalence of types with different but related term substitutions. Consider the standard dependent-function application rule:

$$\frac{\Gamma \vdash e_1 : (x{:}T_1 \to T_2) \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2[e_2/x]}$$

If $e_2 \longrightarrow e_2'$, how do $T_2[e_2/x]$ and $T_2[e_2'/x]$ relate? (An important question when proving preservation!) Subtyping shows that these types are really the same: the first type parallel reduces to the second, and it can be shown that parallel reduction between types implies mutual subtyping—that is, equivalence.

Subtyping brings its own challenges, though. A naïve treatment of subtyping introduces a circularity in the definition of the type system. Existing systems break this circularity by defining judgements in a careful order: first the evaluation relation and the corresponding parallel reduction relation; then a denotational semantics based on the evaluation relation and subtyping based on the denotational semantics; and finally the syntactic type system. Making this carefully sequenced series of definitions hold together requires a long series of tedious lemmas relating evaluation and parallel reduction. The upshot is that existing manifest calculi have taken considerable effort to construct.

We propose here a simpler approach to manifest calculi that greatly simplifies their definition and metatheory. Rather than using subtyping, we define a type conversion relation based on parallel reduction. This avoids the original circularity without resorting to denotational semantics. Indeed, we can use this type conversion to give a completely syntactic account of type soundness—with just a few easy lemmas relating evaluation and parallel reduction. Moreover, eliminating subtyping doesn't fundamentally weaken our approach, since we can define a subtyping relation and prove its soundness *post facto*.

We bring this new technique to bear on $\mathrm{F_H}$, a manifest calculus with parametric polymorphism. Researchers have already studied the *dynamic* enforcement of parametric polymorphism in languages that mix (conventional, un-refined) static and dynamic typing (see Section 6); here we study the *static* enforcement of parametric polymorphism in languages that go beyond conventional static types by adding refinement types and dependent function contracts. Concretely, we offer four main contributions:

1. We devise a simpler approach to manifest contract calculi and apply it to $\mathrm{F_H}$, proving type soundness using straightforward syntactic methods [19].
2. We offer the first operational semantics for *general* refinements, where refinements can apply to any type—not just base types.
3. We prove that $\mathrm{F_H}$ is relationally parametric—establishing that contract checking does not interfere with this desirable property.
4. We define a *post facto* subtyping relation and prove that "upcasts" from subtypes to supertypes always succeed in $\mathrm{F_H}$, i.e., that subtyping is sound.

We begin with some examples in Section 2. We then describe $\mathrm{F_H}$ and prove type soundness in Section 3. We prove parametricity in Section 4 and the upcast lemma in Section 5. We discuss related work in Section 6 and conclude with ideas for future work in Section 7.

2 Examples

Like other manifest calculi, $\mathrm{F_H}$ checks contracts with casts: the cast $\langle T_1 \Rightarrow T_2\rangle^l$ takes a value of type T_1 (the source type) and ensures that it behaves (and is treated) like a T_2 (the target type). The l superscript is a *blame label*, used to differentiate between different casts and identify the source of failures. How we check $\langle T_1 \Rightarrow T_2\rangle^l\, v$ depends on the structure of T_1 and T_2. Checking predicate contracts with casts is easy: if v satisfies the predicate of the target type, the entire application goes to v; if not, then the program aborts, "raising" blame, written $\Uparrow l$. For example, $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\, 5 \longrightarrow^* 5$, since $5 > 0$. But $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\, 0 \longrightarrow^* \Uparrow l$, since $0 \not> 0$. When checking predicate contracts, only the target type matters—the type system guarantees that whatever value we have is well typed at the source type. Checking function contracts is a little trickier: what should $\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}\rangle^l\, v$ do? We can't just open up v and check whether it always returns positives. The solution is to decompose the cast into its parts:

$$\begin{array}{l}\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}\rangle^l\, v \quad \longrightarrow \\ \lambda x{:}\{x{:}\mathsf{Int} \mid x > 0\}.\ (\langle \mathsf{Int} \Rightarrow \{y{:}\mathsf{Int} \mid y > 5\}\rangle^l\, (v\, (\langle \{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int}\rangle^l\, x)))\end{array}$$

Note that the domain cast is contravariant, while the codomain is covariant: the context will be forced by the type system to provide a positive number, so we need to cast the input to an appropriate type for v. (In this example, the contravariant cast $\langle\{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int}\rangle^l$ will always succeed.) After v returns, we run the covariant codomain cast to ensure that v didn't misbehave. So:

$$\begin{aligned}\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}\rangle^l\ (\lambda x{:}\mathsf{Int}.\ x)\, 6 &\longrightarrow^* 6 \\ \langle \cdots \rangle^l\ (\lambda x{:}\mathsf{Int}.\ 0)\, 6 &\longrightarrow^* \Uparrow l \\ \langle \cdots \rangle^l\ (\lambda x{:}\mathsf{Int}.\ 0)\, (\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^{l'}\ 0) &\longrightarrow^* \Uparrow l'\end{aligned}$$

Note that we omitted the case where a cast function is applied to 0. It is an important property of our system that 0 doesn't have type $\{x{:}\mathsf{Int} \mid x > 0\}$!

With these preliminaries out of the way, we can approach our work: a manifest calculus with polymorphism. The standard polymorphic encodings of existential and product types transfer over to $\mathrm{F_H}$ without a problem. Indeed, our dependent functions allow us to go one step further and encode even dependent products such as $(x : \mathsf{Int}) \times \{y{:}\alpha\ \mathsf{List} \mid \mathsf{length}\ y = x\}$, which represents lists paired with their lengths. Let's look at an example combining contracts and polymorphism—an abstract datatype of natural numbers.

$$\begin{aligned}\mathsf{NAT} : \exists\alpha.\ &(\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{iszero} : (\alpha \to \mathsf{Bool})) \times \\ &(\mathsf{pred} : \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \alpha)\end{aligned}$$

(We omit the implementation, a standard Church encoding.) The NAT interface hides our encoding of the naturals behind an existential type, but it also requires that pred is only ever applied to terms of type $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$. Assuming that $\mathsf{iszero}\,v \longrightarrow^* \mathsf{true}$ iff $v = \mathsf{zero}$, we can infer that pred is never given zero as an argument. Consider the following expression, where I is the interface we specified for NAT and we omit the term binding for brevity:

$$\mathsf{unpack}\ \mathsf{NAT} : \exists\alpha.\ I\ \mathsf{as}\ \alpha, \_\ \mathsf{in}\ \mathsf{pred}\,(\langle \alpha \Rightarrow \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}\rangle^l\,\mathsf{zero}) : \alpha$$

The application of pred directly to zero would not be well typed, since $\mathsf{zero} : \alpha$. On the other hand, the cast term is well typed, since we cast zero to the type we need. Naturally, this cast will ultimately raise $\Uparrow l$, because $\mathsf{not}\,(\mathsf{iszero}\,\mathsf{zero}) \longrightarrow^* \mathsf{false}$.

The example so far imposes constraints only on the *use* of the abstract datatype, in particular on the use of pred. To have constraints imposed also on the *implementation* of the abstract data type, consider the extension of the interface with a subtraction operation, sub, and a "less than or equal" predicate, leq. We now have the interface:

$$I' = I \times (\mathsf{leq} : \alpha \to \alpha \to \mathsf{Bool}) \times (\mathsf{sub} : (x{:}\alpha \to \{y{:}\alpha \mid \mathsf{leq}\,y\,x\} \to \{z{:}\alpha \mid \mathsf{leq}\,z\,x\}))$$

The sub function requires that its second argument isn't greater than the first, and it promises to return a result that isn't greater than the first argument.

We get contracts in interfaces by putting casts in the implementations. For example, the contracts on pred and sub are imposed when we "pack up" NAT; we write nat for the implementation type:

$$\mathsf{pack}\ \langle \mathsf{nat}, (\mathsf{zero}, \mathsf{succ}, \mathsf{iszero}, \mathsf{pred}, \mathsf{leq}, \mathsf{sub})\rangle\ \mathsf{as}\ \exists\alpha.\ I'$$

Types and contexts
$T ::= B \mid \alpha \mid x{:}T_1 \to T_2 \mid \forall\alpha.T \mid \{x{:}T \mid e\}$
$\Gamma ::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, \alpha$

Terms
$e ::= x \mid k \mid \mathrm{op}\,(e_1, \ldots, e_n) \mid \lambda x{:}T.\ e \mid \Lambda\alpha.e \mid e_1\, e_2 \mid e\ T \mid$
$\quad \langle T_1 \Rightarrow T_2\rangle^l \mid \Uparrow l \mid \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$
$v ::= k \mid \lambda x{:}T.\ e \mid \Lambda\alpha.e \mid \langle T_1 \Rightarrow T_2\rangle^l$
$r ::= v \mid \Uparrow l$
$E ::= [\,]\ e_2 \mid v_1\ [\,] \mid [\,]\ T \mid \langle\{x{:}T \mid e\}, [\,], v\rangle^l \mid \mathrm{op}(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)$

Fig. 1. Syntax for $\mathrm{F_H}$

where:

$$\begin{aligned} \mathsf{pred} &= \langle \mathsf{nat} \to \mathsf{nat} \Rightarrow \{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \mathsf{nat}\rangle^l\ \mathsf{pred}' \\ \mathsf{sub} &= \langle \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \Rightarrow x{:}\mathsf{nat} \to \{y{:}\mathsf{nat} \mid \mathsf{leq}\ y\ x\} \to \{z{:}\mathsf{nat} \mid \mathsf{leq}\ z\ x\}\rangle^l\ \mathsf{sub}' \end{aligned}$$

That is, the existential type dictates that we must pack up *cast* versions of our implementations, pred' and sub'. Note, however, that the cast on pred' will never actually check anything at runtime: if we unfold the domain contract contravariantly, we see that $\langle\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \Rightarrow \mathsf{nat}\rangle^l$ is a no-op. Instead, clients of NAT can only call pred with terms that are typed at $\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$, i.e., by checking that values are nonzero with a cast into pred's input type. The story is the same for the contract on sub's second argument—the contravariant cast won't actually check anything. The codomain contract on sub, however, could fail if sub' mis-implemented subtraction.

We can sum up the situation for contracts in interfaces as follows: the positive parts of the interface type are checked and can raise blame—these parts are the responsibility of the implementation; the negative parts of the interface type are not checked by the implementation—clients must check these themselves before calling functions from the ADT. Distributing obligations in this way recalls Findler and Felleisen's seminal idea of client and server blame [5].

3 Defining $\mathrm{F_H}$

The syntax of $\mathrm{F_H}$ is given in Figure 1. For unrefined types we have: base types B, which must include Bool; type variables α; dependent function types $x{:}T_1 \to T_2$ where x is bound in T_2; and universal types $\forall\alpha.T$, where α is bound in T. Aside from dependency in function types, these are just the types of the standard polymorphic lambda calculus. As usual, we write $T_1 \to T_2$ for $x{:}T_1 \to T_2$ when x does not appear free in T_2. We also have predicate contracts, or *refinement types*, written $\{x{:}T \mid e\}$. Conceptually, $\{x{:}T \mid e\}$ denotes values v of type T for which $e[v/x]$ reduces to true. For each B, we fix a set $\mathcal{K}_B$ of the constants in that type; we require our typing rules for constants and our typing and evaluation rules for operations to respect this set. We also require that $\mathcal{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

In the syntax of terms, the first line is standard for a call-by-value polymorphic language: variables, constants, several monomorphic first-order operations op (i.e., destructors of one or more base-type arguments), term and type abstractions, and term and type applications. The second line offers the standard constructs of a manifest contract calculus [6, 7, 10], with a few alterations, discussed below.

Casts are the distinguishing feature of manifest contract calculi. When applied to a value of type T_1, the cast $\langle T_1 \Rightarrow T_2\rangle^l$ ensures that its argument behaves—and is treated—like a value of type T_2. When a cast detects a problem, it raises blame, a label-indexed uncatchable exception written $\Uparrow l$. The label l allows us to trace blame back to a specific cast. (While our labels here are drawn from an arbitrary set, in practice l will refer to a source-code location.) Finally, we use active checks $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$ to support a small-step semantics for checking casts into refinement types. In an active check, $\{x{:}T \mid e_1\}$ is the refinement being checked, e_2 is the current state of checking, and v is the value being checked. The type in the first position of an active check isn't necessary for the operational semantics, but we keep it around as a technical aid to type soundness. If checking succeeds, the check will return v; if checking fails, the check will blame its label, raising $\Uparrow l$. Active checks and blame are not intended to occur in source programs—they are runtime devices. (In a real programming language based on this calculus, casts will probably not appear explicitly either, but will be inserted by an elaboration phase. The details of this process are beyond the scope of the present work.)

The values in $\mathrm{F_H}$ are constants, term and type abstractions, and casts. We also define *results*, which are either values or blame. (Type soundness—a consequence of Theorems 2 and 3 below—will show that evaluation produces a result, but not necessarily a value.) In some earlier work [7, 8], casts between function types applied to values were themselves considered values. We make the other choice here: excluding applications from the possible syntactic forms of values simplifies our inversion lemmas.

There are two notable features relative to existing manifest calculi: first, *any* type (even a refinement type) can be refined, not just base types (as in [6–8, 10, 12]); second, the third part of the active check form $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$ can be any value, not just a constant. Both of these changes are motivated by the introduction of polymorphism. In particular, to support refinement of type variables we must allow refinements of *all* types, since any type can be substituted in for a variable.

Operational semantics

The call-by-value operational semantics in Figure 2 are given as a small-step relation, split into two sub-relations: one for reductions ($\rightsquigarrow$) and one for congruence and blame lifting ($\longrightarrow$).

The latter relation is standard. The E_Reduce rule lifts $\rightsquigarrow$ reductions into $\longrightarrow$; the E_Compat rule turns $\longrightarrow$ into a congruence over our evaluation contexts; and the E_Blame rule lifts blame, treating it as an uncatchable exception. The reduction relation $\rightsquigarrow$ is more interesting. There are four different kinds of

Reduction rules $\boxed{e_1 \rightsquigarrow e_2}$

$$
\begin{array}{rll}
\mathrm{op}\,(v_1, \ldots, v_n) & \rightsquigarrow [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n) & \text{E\_Op} \\
(\lambda x{:}T_1.\ e_{12})\, v_2 & \rightsquigarrow e_{12}[v_2/x] & \text{E\_Beta} \\
(\Lambda\alpha.e)\ T & \rightsquigarrow e[T/\alpha] & \text{E\_TBeta}
\end{array}
$$

$$
\begin{array}{rll}
\langle T \Rightarrow T\rangle^l\, v & \rightsquigarrow v & \text{E\_Refl} \\
\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v & \rightsquigarrow & \text{E\_Fun} \\
\multicolumn{2}{l}{\quad \lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)))} & \\
& & \text{when } x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22} \\
\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\, v & \rightsquigarrow \Lambda\alpha.(\langle T_1 \Rightarrow T_2\rangle^l\,(v\ \alpha)) & \text{E\_Forall} \\
& & \text{when } \forall\alpha.T_1 \neq \forall\alpha.T_2
\end{array}
$$

$$
\begin{array}{rll}
\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v & \rightsquigarrow \langle T_1 \Rightarrow T_2\rangle^l\, v & \text{E\_Forget} \\
& \multicolumn{2}{r}{\text{when } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\}} \\
\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v & \rightsquigarrow \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\,(\langle T_1 \Rightarrow T_2\rangle^l\, v) & \text{E\_PreCheck} \\
& & \text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T' \mid e'\} \\
\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v & \rightsquigarrow \langle \{x{:}T \mid e\}, e[v/x], v\rangle^l & \text{E\_Check}
\end{array}
$$

$$
\begin{array}{rll}
\langle \{x{:}T \mid e\}, \mathsf{true}, v\rangle^l & \rightsquigarrow v & \text{E\_OK} \\
\langle \{x{:}T \mid e\}, \mathsf{false}, v\rangle^l & \rightsquigarrow\ \Uparrow l & \text{E\_Fail}
\end{array}
$$

Evaluation rules $\boxed{e_1 \longrightarrow e_2}$

$$
\frac{e_1 \rightsquigarrow e_2}{e_1 \longrightarrow e_2}\ \text{E\_Reduce} \qquad \frac{e_1 \longrightarrow e_2}{E\,[e_1] \longrightarrow E\,[e_2]}\ \text{E\_Compat} \qquad \frac{}{E\,[\Uparrow l] \longrightarrow\ \Uparrow l}\ \text{E\_Blame}
$$

Fig. 2. Operational semantics

reductions: the standard lambda calculus reductions, structural cast reductions, cast staging reductions, and checking reductions.

The E_Beta, and E_TBeta rules should need no explanation—these are the standard call-by-value polymorphic lambda calculus reductions. The E_Op rule uses a denotation function $[\![-]\!]$ to give meaning to our first-order operations.

The E_Refl, E_Fun, and E_Forall rules are structural cast reductions. E_Refl eliminates a cast from a type to itself; intuitively, such a cast should always succeed anyway. (We discuss this rule more in Section 4.) When a cast between function types is applied to a value v, the E_Fun rule produces a new lambda, wrapping v with a contravariant cast on the domain and covariant cast on the codomain. The extra substitution in the left-hand side of the codomain cast may seem suspicious, but in fact the rule must be this way in order for type preservation to hold (see [7] for an explanation). The E_Forall rule is similar to E_Fun, generating a type abstraction with the necessary covariant cast. Side conditions on E_Forall and E_Fun ensure that these rules apply only when E_Refl doesn't.

The E_Forget, E_PreCheck, and E_Check rules are cast-staging reductions, breaking a complex cast down to a series of simpler casts and checks. All of these rules require that the left- and right-hand sides of the cast be different—if they are the same, then E_Refl applies. The E_Forget rule strips a layer of refinement off the left-hand side; in addition to requiring that the left- and right-hand sides are different, the preconditions require that the right-hand side isn't a refinement of the left-hand side. The E_PreCheck rule breaks a cast into two parts: one that checks exactly one level of refinement and another that checks the remaining parts. We only apply this rule when the two sides of the cast are different and when the left-hand side isn't a refinement. The E_Check rule applies when the right-hand side refines the left-hand side; it takes the cast value and checks that it satisfies the right-hand side. (We don't have to check the left-hand side, since that's the type we're casting *from.*)

Before explaining how these rules interact in general, we offer a few examples. First, here is a reduction using E_Check, E_Compat, E_Op, and E_OK:

$$\begin{aligned}\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, 5 &\longrightarrow \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, 5 \geq 0, 5\rangle^l\\ &\longrightarrow \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, \mathsf{true}, 5\rangle^l \longrightarrow 5\end{aligned}$$

A failed check will work the same way until the last reduction, which will use E_Fail rather than E_OK:

$$\begin{aligned}\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, (-1) &\longrightarrow \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, -1 \geq 0, -1\rangle^l\\ &\longrightarrow \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, \mathsf{false}, -1\rangle^l \longrightarrow \Uparrow l\end{aligned}$$

Notice that the blame label comes from the cast that failed. Here is a similar reduction that needs some staging, using E_Forget followed by the first reduction we gave:

$$\begin{aligned}\langle \{x{:}\mathsf{Int} \mid x = 5\} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, 5 &\longrightarrow \langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, 5\\ &\longrightarrow \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, 5 \geq 0, 5\rangle^l \longrightarrow^* 5\end{aligned}$$

There are two cases where we need to use E_PreCheck. First, when multiple refinements are involved:

$$\begin{array}{r}\langle \mathsf{Int} \Rightarrow \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}\rangle^l\, 5 \longrightarrow\\ \langle \{y{:}\mathsf{Int} \mid y \geq 0\} \Rightarrow \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}\rangle^l\, (\langle \mathsf{Int} \Rightarrow \{y{:}\mathsf{Int} \mid y \geq 0\}\rangle^l\, 5) \longrightarrow^*\\ \langle \{y{:}\mathsf{Int} \mid y \geq 0\} \Rightarrow \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}\rangle^l\, 5 \longrightarrow\\ \langle \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}, 5 = 5, 5\rangle^l \longrightarrow^*\\ 5\end{array}$$

Second, when casting a function or universal type into a refinement of a *different* function or universal type.

$$\begin{array}{l}\langle \mathsf{Bool} \to \{x{:}\mathsf{Bool} \mid x\} \Rightarrow \{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\ \mathsf{true} = f\ \mathsf{false}\}\rangle^l\, v \longrightarrow\\ \quad \langle \mathsf{Bool} \to \mathsf{Bool} \Rightarrow \{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\ \mathsf{true} = f\ \mathsf{false}\}\rangle^l\\ \quad\quad (\langle \mathsf{Bool} \to \{x{:}\mathsf{Bool} \mid x\} \Rightarrow \mathsf{Bool} \to \mathsf{Bool}\rangle^l\, v)\end{array}$$

E_Refl is necessary for simple cases, like $\langle \mathsf{Int} \Rightarrow \mathsf{Int}\rangle^l\, 5 \longrightarrow 5$. Hopefully, such a silly cast would never be written, but it could arise as a result of E_Fun or E_Forall. (We also need E_Refl in our proof of parametricity; see Section 4.)

Cast evaluation follows a regular schema:

$$\textsc{Refl} \mid (\textsc{Forget}^* \; (\textsc{Refl} \mid (\textsc{PreCheck}^* \; (\textsc{Refl} \mid \textsc{Fun} \mid \textsc{Forall})? \; \textsc{Check}^*)))$$

Let's consider the cast $\langle T_1 \Rightarrow T_2 \rangle^l \, v$. To simplify the following discussion, we define $\mathrm{unref}(T)$ as T without any outer refinements (though refinements on, e.g., the domain of a function would be unaffected); we write $\mathrm{unref}_n(T)$ when we remove only the n outermost refinements:

$$\mathrm{unref}(T) = \begin{cases} \mathrm{unref}(T') & \text{if } T = \{x{:}T' \mid e\} \\ T & \text{otherwise} \end{cases}$$

First, if $T_1 = T_2$, we can apply E_Refl and be done with it. If that doesn't work, we'll reduce by E_Forget until the left-hand side doesn't have any refinements. (N.B. we may not have to make any of these reductions.) Either all of the refinements will be stripped away from the source type, or E_Refl eventually applies and the entire cast disappears. Assuming E_Refl doesn't apply, we now have $\langle \mathrm{unref}(T_1) \Rightarrow T_2 \rangle^l \, v$. Next, we apply E_PreCheck until the cast is completely decomposed into one-step casts, once for each refinement in T_2:

$$\begin{aligned} &\langle \mathrm{unref}_1(T_2) \Rightarrow T_2 \rangle^l (\langle \mathrm{unref}_2(T_2) \Rightarrow \mathrm{unref}_1(T_2) \rangle^l \\ &\qquad\qquad (\ldots (\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l \, v) \ldots)) \end{aligned}$$

As our next step, we apply whichever structural cast rule applies to $\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l \, v$, one of E_Refl, E_Fun, or E_Forall. Now all that remains are some number of refinement checks, which can be dispatched by the E_Check rule (and other rules, of course, during the predicate checks themselves).

Static typing

The type system comprises three mutually recursive judgments: context well formedness, type well formedness, and term well typing. The rules for contexts and types are unsurprising. The rules for terms are mostly standard. First, the T_App rule is dependent, to account for dependent function types. The T_Cast rule is standard for manifest calculi, allowing casts between compatibly structured well formed types. Compatibility of type structures is defined in Figure 4; in short, compatible types erase to identical simple type skeletons. Note that we assign casts a non-dependent function type. The T_Op rule uses the ty function to assign (possibly dependent) monomorphic first-order types to our operations; we require that ty(op) and $[\![\mathrm{op}]\!]$ agree.

Some of the typing rules—T_Check, T_Blame, T_Exact, T_Forget, and T_Conv—are "runtime only". We don't expect to use these rules to type check source programs, but we need them to guarantee preservation. Note that the conclusions of these rules use a context Γ, but their premises don't use Γ at all. Even though runtime terms and their typing rules should only ever occur in an empty context, the T_App rule substitutes terms into types—so a runtime term could end up under a binder. We therefore allow the runtime typing rules

Context well formedness $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ \textsc{WF\_Empty} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T}{\vdash \Gamma, x{:}T}\ \textsc{WF\_ExtendVar} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \alpha}\ \textsc{WF\_ExtendTVar}$$

Type well formedness $\boxed{\Gamma \vdash T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash B}\ \textsc{WF\_Base} \qquad \frac{\vdash \Gamma \quad \alpha \in \Gamma}{\Gamma \vdash \alpha}\ \textsc{WF\_TVar} \qquad \frac{\Gamma, \alpha \vdash T}{\Gamma \vdash \forall \alpha. T}\ \textsc{WF\_Forall}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \to T_2}\ \textsc{WF\_Fun} \qquad \frac{\Gamma \vdash T \quad \Gamma, x{:}T \vdash e : \mathsf{Bool}}{\Gamma \vdash \{x{:}T \mid e\}}\ \textsc{WF\_Refine}$$

Term typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T\_Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \textsc{T\_Const} \qquad \frac{\emptyset \vdash T \quad \vdash \Gamma}{\Gamma \vdash \Uparrow l : T}\ \textsc{T\_Blame}$$

$$\frac{\Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : x{:}T_1 \to T_2}\ \textsc{T\_Abs} \qquad \frac{\Gamma \vdash e_1 : (x{:}T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2[e_2/x]}\ \textsc{T\_App}$$

$$\frac{\begin{array}{c}\vdash \Gamma \qquad \mathsf{ty}(\mathrm{op}) = x_1 : T_1 \to \ \ldots\ \to x_n : T_n \to T \\ \Gamma \vdash e_i[e_1/x_1, \ldots, e_{i-1}/x_{i-1}] : T_i[e_1/x_1, \ldots, e_{i-1}/x_{i-1}]\end{array}}{\Gamma \vdash \mathsf{op}\,(e_1, \ldots, e_n) : T[e_1/x_1, \ldots, e_n/x_n]}\ \textsc{T\_Op}$$

$$\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. T}\ \textsc{T\_TAbs} \qquad \frac{\Gamma \vdash e_1 : \forall \alpha. T \quad \Gamma \vdash T_2}{\Gamma \vdash e_1\ T_2 : T[T_2/\alpha]}\ \textsc{T\_TApp}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2 \quad T_1 \parallel T_2}{\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T_1 \to T_2}\ \textsc{T\_Cast}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l : \{x{:}T \mid e_1\}}\ \textsc{T\_Check}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash e : T \quad \emptyset \vdash T' \quad T \equiv T'}{\Gamma \vdash e : T'}\ \textsc{T\_Conv} \qquad \frac{\emptyset \vdash v : \{x{:}T \mid e\} \quad \vdash \Gamma}{\Gamma \vdash v : T}\ \textsc{T\_Forget}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \emptyset \vdash \{x{:}T \mid e\} \quad e[v/x] \longrightarrow^* \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}}\ \textsc{T\_Exact}$$

Fig. 3. Typing rules

Type compatibility $\boxed{T_1 \parallel T_2}$

$$\frac{}{T \parallel T}\ \text{C\_Refl} \qquad \frac{T_1 \parallel T_2}{\{x{:}T_1 \mid e\} \parallel T_2}\ \text{C\_RefineL} \qquad \frac{T_1 \parallel T_2}{T_1 \parallel \{x{:}T_2 \mid e\}}\ \text{C\_RefineR}$$

$$\frac{T_{11} \parallel T_{21} \quad T_{12} \parallel T_{22}}{x{:}T_{11} \to T_{12} \parallel x{:}T_{21} \to T_{22}}\ \text{C\_Fun} \qquad \frac{T_1 \parallel T_2}{\forall\alpha.T_1 \parallel \forall\alpha.T_2}\ \text{C\_Forall}$$

Fig. 4. Type compatibility

to apply in any well formed context, so long as the terms they type check are closed. The T_Blame rule allows us to give any type to blame—this is necessary for preservation. The T_Check rule types an active check, $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$. Such a term arises when a term like $\langle T \Rightarrow \{x{:}T \mid e_1\}\rangle^l\ v$ reduces by E_Check. The premises of the rule are all intuitive except for $e_1[v/x] \longrightarrow^* e_2$, which is necessary to avoid nonsensical terms like $\langle\{x{:}T \mid x \geq 0\}, \mathsf{true}, -1\rangle^l$, where the wrong predicate gets checked. The T_Exact rule allows us to retype a closed value of type T at $\{x{:}T \mid e\}$ if $e[v/x] \longrightarrow^* \mathsf{true}$. This typing rule guarantees type preservation for E_OK: $\langle\{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \longrightarrow v$. If the active check was well typed, then we know that $e_1[v/x] \longrightarrow^* \mathsf{true}$, so T_Exact applies. Finally, the T_Conv rule allows us to retype expressions at convertible types: if $\emptyset \vdash e : T$ and $T \equiv T'$, then $\emptyset \vdash e : T'$ (or in any well formed context Γ). We define $\equiv$ as the symmetric, transitive closure of call-by-value respecting parallel reduction, which we write $\Rrightarrow$. The T_Conv rule is necessary to prove preservation in the case where $e_1\ e_2 \longrightarrow e_1\ e_2'$. Why? The first term is typed at $T_2[e_2/x]$ (by T_App), but reapplying T_App types the second term at $T_2[e_2'/x]$. Conveniently, $T_2[e_2/x] \Rrightarrow T_2[e_2'/x]$, so the two are convertible if we take parallel reduction as our type conversion. Naturally, we have to take the transitive closure so we can string together conversion derivations. We take the symmetric closure, since it is easier for us to work with an equivalence. In previous work, subtyping is used instead of the $\equiv$ relation; one of our contributions is the insight that subtyping—with its accompanying metatheoretical complications—is not an essential component of manifest calculi.

We define type compatibility and a few metatheoretically useful operators in Figure 4.

Lemma 1 (Canonical forms). *If $\emptyset \vdash v : T$, then:*

1. *If* $\mathrm{unref}(T) = B$ *then* $v = k \in \mathcal{K}_B$ *for some* v
2. *If* $\mathrm{unref}(T) = x{:}T_1 \to T_2$ *then* v *is*
   (a) $\lambda x{:}T_1'.\ e_{12}$ *and* $T_1' \equiv T_1$ *for some* x, T_1' *and* e_{12}*, or*
   (b) $\langle T_1' \Rightarrow T_2'\rangle^l$ *and* $T_1' \equiv T_1$ *and* $T_2' \equiv T_2$ *for some* T_1', T_2'*, and* l
3. *If* $\mathrm{unref}(T) = \forall\alpha.T'$ *then* v *is* $\Lambda\alpha.v'$ *for some* v'.

Theorem 2 (Progress). *If $\emptyset \vdash e : T$, then either $e \longrightarrow e'$ or e is a result.*

Theorem 3 (Preservation). *If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.*

Closed terms $\boxed{r_1 \sim r_2 : T;\theta;\delta \text{ and } e_1 \simeq e_2 : T;\theta;\delta}$

$$\begin{array}{rcl}
k \sim k : B;\theta;\delta & \iff & k \in \mathcal{K}_B \\
v_1 \sim v_2 : \alpha;\theta;\delta & \iff & \exists R\, T_1\, T_2,\ \alpha \mapsto R,\, T_1,\, T_2 \in \theta \wedge v_1\ R\ v_2 \\
v_1 \sim v_2 : (x{:}T_1 \to T_2);\theta;\delta & \iff & \forall v_1' \sim v_2' : T_1;\theta;\delta,\ v_1\, v_1' \simeq v_2\, v_2' : T_2;\theta;\delta[v_1', v_2'/x] \\
v_1 \sim v_2 : \forall\alpha.T;\theta;\delta & \iff & \forall R\, T_1\, T_2,\ v_1\ T_1 \simeq v_2\ T_2 : T;\theta[\alpha \mapsto R,\, T_1,\, T_2];\delta \\
v_1 \sim v_2 : \{x{:}T \mid e\};\theta;\delta & \iff & v_1 \sim v_2 : T;\theta;\delta \wedge \\
 & & \theta_1(\delta_1(e))[v_1/x] \longrightarrow^* \mathsf{true} \wedge \theta_2(\delta_2(e))[v_2/x] \longrightarrow^* \mathsf{true} \\
\Uparrow l \sim \Uparrow l : T;\theta;\delta & & \\
 & & \\
e_1 \simeq e_2 : T;\theta;\delta & \iff & \exists r_1 r_2, e_1 \longrightarrow^* r_1 \wedge e_2 \longrightarrow^* r_2 \wedge r_1 \sim r_2 : T;\theta;\delta
\end{array}$$

Types $\boxed{T_1 \simeq T_2 : *;\theta;\delta}$

$$\begin{array}{rcl}
B \simeq B : *;\theta;\delta & & \\
\alpha \simeq \alpha : *;\theta;\delta & & \\
x{:}T_{11} \to T_{12} \simeq x{:}T_{21} \to T_{22} : *;\theta;\delta & \iff & T_{11} \simeq T_{21} : *;\theta;\delta \wedge \\
 & & \forall v_1 \sim v_2 : T_{11};\theta;\delta, \\
 & & \quad T_{12} \simeq T_{22} : *;\theta;\delta[v_1, v_2/x] \\
\forall\alpha.T_1 \simeq \forall\alpha.T_2 : *;\theta;\delta & \iff & \forall R\, T_1'\, T_2',\ T_1 \simeq T_2 : *;\theta[\alpha \mapsto R,\, T_1',\, T_2'];\delta \\
\{x{:}T_1 \mid e_1\} \simeq \{x{:}T_2 \mid e_2\} : *;\theta;\delta & \iff & T_1 \simeq T_2 : *;\theta;\delta \wedge \\
 & & \forall v_1 \sim v_2 : T_1;\theta;\delta,\ \theta_1(\delta_1(e_1))[v_1/x] \simeq \theta_2(\delta_2(e_2))[v_2/x] : \mathsf{Bool};\theta;\delta
\end{array}$$

Open terms and types $\boxed{\Gamma \vdash \theta;\delta \text{ and } \Gamma \vdash e_1 \simeq e_2 : T \text{ and } \Gamma \vdash T_1 \simeq T_2 : *}$

$$\begin{array}{rcl}
\Gamma \vdash \theta;\delta & \iff & \forall x{:}T \in \Gamma,\ \theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T;\theta;\delta \wedge \\
 & & \forall\alpha \in \Gamma, \exists R\, T_1\, T_2,\ \alpha \mapsto R,\, T_1,\, T_2 \in \theta \\
\Gamma \vdash e_1 \simeq e_2 : T & \iff & \forall \Gamma \vdash \theta;\delta,\ \theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : T;\theta;\delta \\
\Gamma \vdash T_1 \simeq T_2 : * & \iff & \forall \Gamma \vdash \theta;\delta,\ T_1 \simeq T_2 : *;\theta;\delta
\end{array}$$

Fig. 5. The logical relation for parametricity

Requiring standard weakening, substitution, and inversion lemmas, the syntactic proof of type soundness is straightforward. It is easy to restrict $\mathrm{F_H}$ to a simply typed calculus with a similar type soundness proof.

4 Parametricity

We prove relational parametricity for two reasons: (1) it gives us powerful reasoning techniques such as free theorems [17], and (2) it indicates that contracts don't interfere with type abstraction. Our proof is standard: we define a (syntactic) logical relation where each type is interpreted as a relation on terms and the relation at type variables is given as a parameter. In the next section, we will define a subtype relation and show that an upcast—a cast whose source type is a subtype of the target type—is logically related to the identity function. Since our logical relation is an adequate congruence, it is contained in contextual equivalence. Therefore, upcasts are *contextually equivalent* to the identity and can be eliminated without changing the meaning of a program.

We begin by defining two relations: $r_1 \sim r_2 : T;\theta;\delta$ relates closed results, defined by induction on types; $e_1 \simeq e_2 : T;\theta;\delta$ relates closed expressions which evaluate results in the first relation. The definitions are shown in Figure 5.[1] Both relations have three indices: a type T, a substitution θ for type variables, and a substitution δ for term variables. A type substitution θ, which gives the interpretation of free type variables in T, maps a type variable to a triple (R, T_1, T_2) comprising a binary relation R on terms and two closed types T_1 and T_2. We require that R be closed under parallel reduction (the $\Rrightarrow$ relation). A term substitution δ maps from variables to pairs of closed values. We write θ_i $(i = 1, 2)$ for a substitution that maps a type variable α to T_i where $\theta(\alpha) = (R, T_1, T_2)$. We denote projections δ_i similarly.

With these definitions out of the way, the term relation is mostly straightforward. First, $\Uparrow l$ is related to itself at every type. A base type B gives the identity relation on $\mathcal{K}_B$, the set of constants of type B. A type variable α simply uses the relation assumed in the substitution θ. Related functions map related arguments to related results. Type abstractions are related when their bodies are parametric in the interpretation of the type variable. Finally, two values are related at a refinement type when they are related at the underlying type and both satisfy the predicate; here, the predicate e gets closed by applying the substitutions. The $\sim$ relation on results is extended to the relation $\simeq$ on closed terms in a straightforward manner: terms are related if and only if they both terminate at related results. We extend the relation to open terms, written $\Gamma \vdash e_1 \simeq e_2 : T$, relating open terms that are related when closed by any "Γ-respecting" pair of substitutions θ and δ (written $\Gamma \vdash \theta; \delta$, also defined in Figure 5).

To show that a (well-typed) cast is logically related to itself, we also need a relation on types $T_1 \simeq T_2 : *; \theta; \delta$; we define this relation in Figure 5. We use the logical relation on terms to handle the arguments of function types and refinement types. Note that T_1 and T_2 are not necessarily closed; terms in refinement types, which should be related at Bool, are closed by applying substitutions. In the function/refinement type cases, the relation on a smaller type is universally quantified over logically related values. There are two choices of the type at which they should be related (for example, the second line of the function type case could change T_{11} to T_{21}), but it does not really matter which to choose since they are related types. Here, we have chosen the type from the left-hand side; in our proof, we justify this choice by proving a "type exchange" lemma that lets us replace a type index T_1 in the term relation by T_2 when $T_1 \simeq T_2 : *$. Finally, we lift our type relation to open terms: we write $\Gamma \vdash T_1 \simeq T_2 : *$ when two types are equivalent for any Γ-respecting substitutions.

It is worth discussing a few points peculiar to our formulation. First, we allow any relation on terms closed under parallel reduction to be used in θ; terms related at T need not be well typed at T. The standard formulation of a logical relation is well typed throughout, requiring that the relation R in every triple be well typed, only relating values of type T_1 to values of type T_2 (e.g., [14]). We have two motivations for leaving our relations untyped. First, functions of type

[1] To save space, we write $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ separately instead of manually adding such a clause for each type.

$x{:}T_1 \to T_2$ must map related values ($v_1 \sim v_2 : T_1$) to related results...but at what type? While $T_2[v_1/x]$ and $T_2[v_2/x]$ are related in our type logical relation, terms that are well typed at one type won't necessarily be well typed at the other. Second, we prove in Section 5 that upcasts have no effect: if $T_1 <: T_2$, then $\langle T_1 \Rightarrow T_2\rangle^l \sim \lambda x{:}T_1.\ x : T_1 \to T_2$. That is, we want a cast $\langle T_1 \Rightarrow T_2\rangle^l$, of type $T_1 \to T_2$, to be related to the identity $\lambda x{:}T_1.\ x$, of type $T_1 \to T_1$: the cast and the identity won't (in general) have the same type. We therefore don't demand that two expressions related at T be well typed at T, and we allow *any* relation to be chosen as R, so long as it is closed under parallel reduction. Another peculiarity is in our treatment of substitutions and type indices. Just as the interpretation of free type variables in the logical relation's type index are kept in a substitution θ, we keep δ as a substitution for the free term variables that can appear in type indices. Keeping this substitution separate avoids a problem in defining the logical relation at function types. Consider a function type $x{:}T_1 \to T_2$: our *logical* relation says that values v_1 and v_2 are related at this type when they take related values to related results, i.e. if $v_1' \sim v_2' : T_1; \theta; \delta$, then we should be able to find $v_1\, v_1' \simeq v_2\, v_2'$. The question here is which type index we should use. If we keep our type indices closed (with respect to term variables), we cannot use T_2 on its own—we have to choose a binding for x! Knowles and Flanagan [10] deal with this problem by introducing the "wedge product" operator, which merges two types—one with v_1' substituted for x and the other with v_2' for x—into one. Instead of substituting eagerly, we put both bindings in δ and apply them when needed—the refinement type case. We think our formulation is more uniform with regard to free term/type variables, since eager substitution is a non-starter for type variables, anyway.

As we developed our proof, we found that the E_Refl rule

$$\langle T \Rightarrow T\rangle^l\, v \rightsquigarrow v$$

is not just a convenient way to skip decomposition of a trivial cast into smaller trivial casts (when T is a polymorphic or dependent function type); E_Refl is, in fact, crucial to obtaining parametricity in our syntactic setting. For example, by parametricity, we expect every value of type $\forall\alpha.\alpha \to \alpha$ to behave the same as the polymorphic identity function. One of the values of this type is $\Lambda\alpha.\langle\alpha \Rightarrow \alpha\rangle^l$. Without E_Refl, however, applying this type abstraction to a compound type, say $\mathsf{Bool} \to \mathsf{Bool}$, and a function f of type $\mathsf{Bool} \to \mathsf{Bool}$ would return, by E_Fun, a value that is syntactically different from f, breaking parametricity![2] With E_Refl, $\langle T \Rightarrow T\rangle^l$ returns the input immediately, regardless of T, just as the identity function. So, this rule is a technical necessity, ensuring that casts containing type variables behave parametrically. (Naturally, the evaluation of well-typed programs never encounters casts with uninstantiated type variables.)

We have relational parametricity—every well-typed term (under Γ) is related to itself for any Γ-respecting substitutions.

[2] Intuitively, we expect the returned value should behave the same as the input, though. Moreover, the subtyping we define is reflexive, so the upcast lemma we prove applies, as well—though, of course, we used E_Refl to prove it!

$$\boxed{\Gamma \vdash T_1 <: T_2}$$

$$\frac{}{\Gamma \vdash B <: B}\ \textsc{S\_Base} \qquad \frac{}{\Gamma \vdash \alpha <: \alpha}\ \textsc{S\_TVar} \qquad \frac{\Gamma, \alpha \vdash T_1 <: T_2}{\Gamma \vdash \forall\alpha.T_1 <: \forall\alpha.T_2}\ \textsc{S\_Forall}$$

$$\frac{\Gamma \vdash T_{21} <: T_{11} \quad \Gamma, x{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] <: T_{22}}{\Gamma \vdash x{:}T_{11} \to T_{12} <: x{:}T_{21} \to T_{22}}\ \textsc{S\_Fun}$$

$$\text{casts}(T) = \begin{cases} \langle T' \Rightarrow \{x{:}T' \mid e\}\rangle^l \circ \text{casts}(T') & \text{if } T = \{x{:}T' \mid e\} \\ \lambda x{:}T.\ x & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{l}\Gamma \vdash \text{unref}(T_1) <: \text{unref}(T_2) \\ \Gamma, x{:}\text{unref}(T_1) \vdash \text{casts}(T_1)\, x \supset \text{casts}(T_2)\,(\langle \text{unref}(T_1) \Rightarrow \text{unref}(T_2)\rangle^l\, x)\end{array}}{\Gamma \vdash T_1 <: T_2}\ \textsc{S\_Refine}$$

$$\boxed{\Gamma \vdash e_1 \supset e_2}$$

$$\frac{\forall \Gamma \vdash \theta; \delta.\ (\exists v.\ \theta_1(\delta_1(e_1)) \longrightarrow^* v)\ \text{implies}\ (\exists v.\ \theta_1(\delta_1(e_2)) \longrightarrow^* v)}{\Gamma \vdash e_1 \supset e_2}\ \textsc{Imp}$$

Fig. 6. Subtyping, implication, and closing substitutions

Theorem 4 (Parametricity).

1. *If* $\Gamma \vdash e : T$ *then* $\Gamma \vdash e \simeq e : T$*, and*
2. *If* $\Gamma \vdash T$ *then* $\Gamma \vdash T \simeq T : *$.

The proof is mostly standard, although—like the proof of semantic type soundness in Greenberg, Pierce, and Weirich [7]—it requires a separate reflexivity lemma for casts, as mentioned above. We make one small disclaimer: we have not completed the standard but tedious proof showing that parallel reduction implies cotermination at similar values, i.e., if $e_1 \Rrightarrow e_2$ and $e_1 \longrightarrow^* r_1$, then $e_2 \longrightarrow^* r_2$ such that $r_1 \Rrightarrow r_2$, and vice versa. We expect that our existing Coq proof of this fact for a similar operational semantics (from [7]) will adapt readily. Note that our proof of type soundness in Section 3 relies on much simpler properties of parallel reduction, which we *have* proved.

5 Subtyping and Upcast Elimination

Knowles and Flanagan [10] define a subtyping relation for their manifest calculus, λ_H, as a primitive notion of the system. Furthermore, they prove that upcast elimination is sound: if $T_1 <: T_2$, then $\langle T_1 \Rightarrow T_2\rangle^l$ is equivalent to the identity function. Upcast elimination is, at heart, an optimization: since the cast can never fail, there is no point in running it. We define a subtyping relation for F_H and prove that upcast elimination is sound. To be clear, the type system of F_H doesn't have subtyping or a subsumption rule at all; we simply show that upcasts are logically related—and therefore contextually equivalent—to the identity.

We define subtyping in Figure 6. Our subtyping rules are similar to those in λ_H. The first three rules are standard. The rule for dependent function types is mostly usual: contravariant on argument types and covariant on return types. Here, we need to be careful about the type of x. Return types T_{12} and T_{22} should be compared under the assumption that x has T_{21}, which is a subtype of the other argument type T_{11} [4]. However, x in T_{12} has a different type, i.e., T_{11}, so we need to insert a cast to keep the subtyping relation well typed—F_H doesn't have subsumption!

Our rule for subtyping of refinements differs substantially from λ_H's, mostly because F_H allows refinements of arbitrary types, while λ_H only refines base types. The S_REFINE rule essentially says T_1 is a subtype of T_2 if (1) T_1 without the (outermost) refinements is a subtype of T_2 without the (outermost) refinements, and (2) for any v of type $\mathrm{unref}(T_1)$, if $\mathrm{casts}(T_1)\, v$ reduces to a value, so does $\mathrm{casts}(T_2)\, \langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\, v$, for any l. The intuition behind the second condition is that, for T_1 to be a subtype of T_2, the predicates in T_1 (combined by conjunction) should be stronger than those in T_2. Recall that $\mathrm{casts}(T)$ is defined in Figure 6 as the composition of casts necessary to cast from $\mathrm{unref}(T)$ to T. So, if application of $\mathrm{casts}(T)$ to a value of $\mathrm{unref}(T)$ does not raise blame, then the value can be typed at T by repeated use of T_EXACT.

If the implication in S_REFINE holds for a value v of type $\mathrm{unref}(T_1)$, then either: (1) v did not pass the checks in $\mathrm{casts}(T_1)$, so this value is not in T_1; or (2) v passed the checks in $\mathrm{casts}(T_1)$ and $\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\, v$ passed all of the checks in $\mathrm{casts}(T_2)$. So, if (1) or (2) hold for all values of type $\mathrm{unref}(T_1)$, then it means that all values of type T_1 can be safely treated as if they had type T_2, i.e., T_1 a subtype of T_2.

Finally, we need a source of closing substitutions to compare the evaluation of the two casts. We use the closing substitutions from the logical relation at T as the source of "values of type T". (Arbitrarily, we take the values and types from the left.) There is a similar situation in the manifest calculi of Knowles and Flanagan [10] and Greenberg, Pierce, and Weirich [7]. They both define a denotational semantics for use in their refinement subtyping rule—but they *need* to do so, in order to avoid a circularity. We have no such issues, and make the decision because it is expedient.

We formulate our implication judgment in terms of cotermination at values rather than cotermination at `true` (as in [7, 10]) because we have to contend with multiple layers of refinement in types—using cotermination at values reduces the amount of predicate bookkeeping we have to do.

Having defined subtyping, we are able to show that upcast elimination is sound.

Lemma 5 (Upcast lemma). *If $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_1$ and $\Gamma \vdash T_2$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l \simeq \lambda x{:}T_1.\ x : T_1 \to T_2$.*

6 Related Work

We discuss the related work in two parts. We first distinguish our work from the untyped contract systems that enforce parametric polymorphism *dynamically*, rather than statically as F_H does; we then discuss how F_H differs from existing manifest contract calculi in greater detail.

Dynamically checked polymorphism

The F_H type system enforces parametricity with type abstractions and type variables, while refinements are dynamically checked. Another line of work omits refinements, seeking instead to dynamically enforce parametricity—typically with some form of sealing (à la Pierce and Sumii [13]).

Guha et al. [9] define contracts with polymorphic signatures, maintaining abstraction with sealed "coffers"; they do not prove parametricity. Matthews and Ahmed [11] prove parametricity for a polymorphic multi-language system with a similar policy. Ahmed et al. [2] prove parametricity for a gradual typing [15] calculus which enforces polymorphism with a set of global runtime seals. Strickland et al. add support for dynamically checked variable-arity polymorphism to Typed Racket [16]. Ahmed et al. [3] define a polymorphic calculus for gradual typing, using local syntactic "barriers" instead of global seals. We believe that it is possible to combine F_H with the barrier calculus of Ahmed et al., yielding a polymorphic blame calculus [18]. We leave this to future work.

Manifest systems

Wadler and Findler [18] gave a simple syntactic account of a calculus combining refinement types and gradual types [15]; they, like us, define subtyping *post facto*, proving theorems similar to the upcast lemma. They do not, however, support dependent function types. Gronski and Flanagan [8] compares non-dependent latent and manifest contract calculi.

Four existing manifest calculi have dependent function types (such as [6, 7, 10, 12]) use subtyping and theorem provers as part of the definition of the type system. All four of these calculi have complicated metatheory. Ou et al. [12] restrict refinements and arguments of dependent functions to a conservative approximation of pure terms; they also place strong requirements on their prover. Knowles and Flanagan [10] as well as Greenberg, Pierce, and Weirich [7] use denotational semantics to give a firm foundation to Flanagan's earlier work [6]. We consider three systems in more detail: Knowles and Flanagan's λ_H (KF) [10]; Greenberg, Pierce, and Weirich's λ_H (GPW) [7]; and F_H. The rest of this subsection addresses the differences between KF, GPW, and F_H.

In Section 1, we discussed in general terms some of the complexity that KF and GPW encountered. What made KF and GPW so complicated? Both systems share the same two impediments in the preservation proof: preservation after active checks and after congruence steps in the argument position of applications.

KF and GPW resolve both of these with subtyping, using a rule like the following for refinements:[3]

$$\frac{\forall \Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma.\ \sigma(e_1) \longrightarrow^* \mathsf{true}\,\text{implies}\,\sigma(e_2) \longrightarrow^* \mathsf{true}}{\Gamma \vdash \{x{:}B \mid e_1\} <: \{x{:}B \mid e_2\}}$$

Subtyping and the requirement that constants be assigned most specific types, —i.e., if $e[k/x] \longrightarrow^* \mathsf{true}$ for $k \in \mathcal{K}_B$ then $\emptyset \vdash \mathsf{ty}\,(k) <: \{x{:}B \mid e\}$—are used to show preservation of active checks. The two systems use subtyping to relate substituted types in different ways. KF use full beta reduction, showing that subtyping is closed under reduction. GPW use call-by-value reduction, showing that subtyping is closed under parallel reduction. Once these two difficulties are resolved, both preservation proofs are standard, given appropriate subtyping inversion lemmas.

So much for subtyping. Why do KF and GPW need denotational semantics? Spelled out pedantically, the subtyping rule above has the following premise:

$$\forall \sigma.\ \Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma\,\text{implies}\,(\sigma(e_1) \longrightarrow^* \mathsf{true}\,\text{implies}\,\sigma(e_2) \longrightarrow^* \mathsf{true})$$

That is, the well formedness of the closing substitution σ is in a negative position. Where do closing substitutions come from? We cannot use the typing judgment itself, as this would be ill-defined: term typing requires subtyping via subsumption; subtyping requires closing substitutions in a negative position via the refinement case; but closing substitutions require typing. We need another source of values: hence, denotational semantics. Both KF and GPW define syntactic term models of types to use as a source of values for closing substitutions, though the specifics differ.

After adding subtyping and denotational semantics, both KF and GPW are well defined and have syntactic proofs of type soundness. But in the process of proving syntactic type soundness, both languages proved semantic soundness theorems:

$$\Gamma \vdash e : T \text{ implies } \forall \Gamma \vdash \sigma,\ \sigma(e) \in [\![\sigma(T)]\!]$$

This theorem suffices for soundness of the language... so why bother with a syntactic proof? In light of this, GPW only proves semantic soundness. The situation in KF and GPW is unsatisfying: the syntactic proof of type soundness motivated subtyping, which motivated denotational semantics, which obviated the need for syntactic proof. Beyond this, the proofs are hard to scale: adding in polymorphism or state is a non-trivial task, since we must—before defining the type system!—construct an appropriate denotational semantics, which itself depends on our evaluation relation.

$\mathrm{F_H}$ solves the problem by avoiding subtyping—which is what forced the presence of closing substitutions and denotational semantics in the first place. The first issue in preservation—that of preserving refinement types after checks have finished—was resolved in KF and GPW with subtyping. We instead resolve it

[3] Readers familiar with the systems will recognize that we've folded the implication judgment into the relevant subtyping rule.

with a runtime rule that allows us to type values with any refinement they satisfy:

$$\frac{\vdash \Gamma \qquad \emptyset \vdash v : T \qquad \emptyset \vdash \{x{:}T \mid e\} \qquad e[v/x] \longrightarrow^* \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}} \quad \text{T\_EXACT}$$

Adding this rule eliminates one use of subtyping as well as the “most-specific type” restriction. If we “bit the bullet” and allowed non-empty contexts in T_EXACT, then we would need to apply a closing substitution to $e[v/x]$ before checking if it reduces to true. But the circularity in subtyping alluded to in Section 1 was caused by closing substitutions; we must avoid them! The second issue in preservation—that of conversion between $T_2[e_2/x]$ and $T_2[e_2'/x]$—can be resolved in a similar fashion. We define another runtime rule that allows us to convert types:

$$\frac{\vdash \Gamma \qquad \emptyset \vdash e : T \qquad \emptyset \vdash T' \qquad T \equiv T'}{\Gamma \vdash e : T'} \quad \text{T\_CONV}$$

The conversion we use, $\equiv$, is defined as the symmetric, transitive closure of CBV-respecting parallel reduction. This is only as much equivalence as we need: if $e_2 \longrightarrow e_2'$, then $T_2[e_2/x] \equiv T_2[e_2'/x]$. These two rules suffice to keep subtyping out of $\mathrm{F_H}$, which in turn avoids denotational semantics.

7 Future Work

We presented a simpler approach to manifest contract calculi, which we applied to construct $\mathrm{F_H}$, a parametrically polymorphic manifest contract calculus. We hope to extend $\mathrm{F_H}$ with barriers for dynamically checked polymorphism [3], and with general recursion and state. (Though we acknowledge that state is a difficult open problem.) We also hope that $\mathrm{F_H}$'s operational semantics and (relatively) simple type system will help developers implement contracts.

Acknowledgments

Stephanie Weirich provided many insights throughout. Jianzhou Zhao's help with parametricity was invaluable; and a conversation about parametricity with Amal Ahmed and Stephanie Weirich was particularly illuminating. This work was supported in part by the National Science Foundation under grant 0915671, *Contracts for Precise Types*, in part by the JSPS Grant-in-Aid for Young Scientists (A) No. 21680002, and in part by the Portuguese Foundation for Science and Technology, POPH - QREN, under grant SFRH / BPD / 46065 / 2008.

References

1. PLT Racket Contracts, http://pre.plt-scheme.org/docs/html/guide/contracts.html

2. Ahmed, A., Findler, R.B., Matthews, J., Wadler, P.: Blame for all. In: Workshop on Script-to-Program Evolution (STOP) (2009)
3. Ahmed, A., Findler, R.B., Siek, J., Wadler, P.: Blame for all. In: Principles of Programming Languages (POPL) (2011)
4. Aspinall, D., Compagnoni, A.: Subtyping dependent types. Theor. Comput. Sci. 266(1-2), 273–309 (Sep 2001)
5. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: International Conference on Functional Programming (ICFP). pp. 48–59 (2002)
6. Flanagan, C.: Hybrid type checking. In: POPL. pp. 245–256 (2006)
7. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: Principles of Programming Languages (POPL) 2010 (2010)
8. Gronski, J., Flanagan, C.: Unifying hybrid types and contracts. In: Trends in Functional Programming (TFP) (2007)
9. Guha, A., Matthews, J., Findler, R.B., Krishnamurthi, S.: Relationally-parametric polymorphic contracts. In: DLS. pp. 29–40 (2007)
10. Knowles, K., Flanagan, C.: Hybrid type checking (2010), to appear in TOPLAS.
11. Matthews, J., Ahmed, A.: Parametric polymorphism through run-time sealing or, theorems for low, low prices! In: European Symposium on Programming. pp. 16–31. Springer-Verlag, Berlin, Heidelberg (2008)
12. Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: IFIP TCS. pp. 437–450 (2004)
13. Pierce, B., Sumii, E.: Relating cryptography and polymorphism (Jul 2000)
14. Pitts, A.M.: Typed operational reasoning. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, chap. 7, pp. 245–289. The MIT Press (2005)
15. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
16. Strickland, T.S., Tobin-Hochstadt, S., Felleisen, M.: Practical variable-arity polymorphism. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems. pp. 32–46. Springer-Verlag, Berlin, Heidelberg (2009)
17. Wadler, P.: Theorems for free! In: Proceedings of ACM Conference on Functional Programming and Computer Architecture (FPCA'89). pp. 347–359. London, UK (Sep 1989)
18. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: European Symposium on Programming (ESOP). pp. 1–16 (2009)
19. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115, 38–94 (1992)