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 abstract types strong contracts, 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 its 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

Disclaimer

This is the long version of our ESOP 2011 subsmission. It contains the proofs in full and some extra discussion, but we haven't spent as long preparing the text.

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, which can take one of two forms: predicate contracts like $\{x{:}\mathsf{Int} \mid x > 0\}$, denoting the positive numbers, and function contracts like $x{:}\mathsf{Int} \rightarrow \{y{:}\mathsf{Int} \mid y \geq x\}$, denoting monotonically increasing functions over the integers.

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 application 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 adopt the manifest approach and endeavor to tame its principal drawback: the complexity of its metatheory.

Subtyping is the source of complexity in the most expressive manifest calculi—those with arbitrary predicates and dependent functions [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 allows us to type n 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? (This is an important question in a proof of 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 that leaves the type system undefined. Existing systems break the circularity by defining their systems 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 and fiddly lemmas relating evaluation and parallel reduction. In short, existing manifest calculi have taken a great deal of effort to construct.

We have developed 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—there is no risk of circularity, obviating the need for denotational semantics. We can use this type conversion to give a completely syntactic account of type soundness—with only a few easy lemmas relating evaluation and parallel reduction. We should add that eliminating subtyping doesn't weaken our approach: it is possible to define a subtyping relation and prove its soundness *post facto*.

We bring this new technique to bear in $\mathrm{F_H}$, a manifest calculus with parametric polymorphism. We should be clear: researchers have already studied the *dynamic* enforcement of parametric polymorphism in languages that mix (conventional, un-refined) static and dynamic typing; we study the *static* enforcement of parametric polymorphism in languages that go beyond conventional static types by adding refinement types and dependent function contracts. We discuss the those systems more in Section 6. Concretely, we offer three 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 prove that $\mathrm{F_H}$ is relationally parametric—establishing that contract checking does not interfere with this desirable property.
3. 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 our technical work 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 $\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{x{:}\mathsf{Int} \mid x > 0\} \rangle^l\, v$

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:

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

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:

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

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

As for polymorphism, the standard 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.

With these preliminaries out of the way, let's look at an example combining contracts and polymorphism—an abstract datatype of natural numbers.

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

(We omit the implementation, a standard Church encoding.) The NAT interface hides our encoding of the naturals behind an existential type, but it also guarantees 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, this term is well typed, since we cast zero to the type we need. This program 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:

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

Types and contexts

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

Terms

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

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

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:

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

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. This pattern is reminiscent of the original idea of client and server blame found in Findler and Felleisen's seminal work [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. 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, first-order operations (on one or more 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. 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 22 and 23 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 con-

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

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

$$
\begin{array}{rlr}
\langle T \Rightarrow T\rangle^l\ v & \rightsquigarrow v & \textsc{E\_Refl} \\
\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\ v & \rightsquigarrow & \textsc{E\_Fun} \\
\multicolumn{3}{l}{\quad \lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\ x/x] \Rightarrow T_{22}\rangle^l\ (v\ (\langle T_{21} \Rightarrow T_{11}\rangle^l\ x)))} \\
\multicolumn{3}{r}{\text{when } x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22}} \\
\langle \forall\alpha.\ T_1 \Rightarrow \forall\alpha.\ T_2\rangle^l\ v & \rightsquigarrow \Lambda\alpha.\ (\langle T_1 \Rightarrow T_2\rangle^l\ (v\ \alpha)) & \textsc{E\_Forall} \\
\multicolumn{3}{r}{\text{when } \forall\alpha.\ T_1 \neq \forall\alpha.\ T_2}
\end{array}
$$

$$
\begin{array}{rlr}
\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\ v & \rightsquigarrow \langle T_1 \Rightarrow T_2\rangle^l\ v & \textsc{E\_Forget} \\
\multicolumn{3}{r}{\text{when } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\}} \\
\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\ v & \rightsquigarrow \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\ (\langle T_1 \Rightarrow T_2\rangle^l\ v) & \textsc{E\_PreCheck} \\
\multicolumn{3}{r}{\text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T' \mid e'\}} \\
\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\ v & \rightsquigarrow \langle \{x{:}T \mid e\}, e[v/x], v\rangle^l & \textsc{E\_Check}
\end{array}
$$

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

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

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

Fig. 2. Operational semantics

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

The E_Op, E_Beta, and E_TBeta rules should need no explanation—these are the standard call-by-value polymorphic lambda calculus reductions.

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. This rule applies 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's 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's 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:

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

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

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

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 $\langle T_1 \Rightarrow T_2 \rangle^l \, v$. 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:

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

(We write unref_n to strip away exactly n levels of refinements.) 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).

The E_Refl rule merits some more discussion. On the face of it, a cast $\langle T \Rightarrow T \rangle^l$ seems like it can't do anything: any value it applies must have already been typed T, so what could go wrong during any checks? We might not worry on account of the upcast theorem of Knowles and Flanagan [10]: they show that any cast $\langle T_1 \Rightarrow T_2 \rangle^l$ where $T_1 <: T_2$ is contextually equivalent to the identity function—it has no effect. Since their subtyping is reflexive, this theorem applies to a cast like $\langle T \Rightarrow T \rangle^l$. But their upcast theorem relies on a typing judgment with subtyping built in. To be fair, their subtyping is sound with respect to the denotational semantics; they're using subtyping to build that semantics into their type system. As we are about to show, we make do with fewer assumptions in our type system; however, we haven't been able to get around the need for the E_Refl rule. Knowles and Flanagan baked some semantic facts into their type system; we baked—fewer!—semantic facts into our operational semantics.

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Typing rules

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 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 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 $\Rrightarrow$. This 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. 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. We define $\mathrm{casts}(T)$ as the composition of casts necessary to cast from $\mathrm{unref}(T)$ to T.

Lemma 1 (Determinism). *If $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

Proof. By case analysis for $\rightsquigarrow$ and induction on $e \longrightarrow e_1$.

Cases for $\rightsquigarrow$:

(E_Op): Immediate; no other rule applies.

(E_Beta): Immediate; no other rule applies.

(E_TBeta): Immediate; no other rule applies.

(E_Refl): E_Forget doesn't apply, since it requires $T_2 \neq \{x{:}T_1 \mid e\}$. E_Check doesn't apply because it requires $T \neq \{x{:}T \mid e\}$. E_PreCheck doesn't apply because it requires that T_1 isn't a refinement and that T_2 is refinement—which means that $T_1 \neq T_2$. Neither E_Fun nor E_Forall can apply, since they require that the types are different as well.

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

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

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

Operations on refinements

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

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

Fig. 4. Operations on types

(E_Forget): E_Check doesn't apply since T_2 isn't a refinement. E_PreCheck doesn't apply because $\{x{:}T_1 \mid e_1\}$ is a refinement.
(E_PreCheck): E_Check can't apply because T_1 is different from T_2.
(E_Check): No other rules for working on refinements remain.
(E_Fun): E_Forall can't apply, because it only works on universal types.
(E_Forall): All other cast rules have already been considered.
(E_OK): Fail can't apply, because $\mathsf{true} \neq \mathsf{false}$.
(E_Fail): All other active check rules have already considered.

Cases for $\longrightarrow$:

(E_Reduce): We already know that $\rightsquigarrow$ is deterministic. Now observe that terms that are subject to reduction rules (uniquely) factor into evaluation contexts where the subterm is a value, E_Compat and E_Blame couldn't have been used.
(E_Compat): Terms factor into evaluation contexts and subterms deterministically. What's more, if the subterm steps it can't be blame, so E_Blame couldn't have been used. By the IH, $e_1 \longrightarrow e_2$ is deterministic as well.
(E_Blame): All other rules have already been considered.

Lemma 2 (Substitutivity of parallel reduction). *If $e_1' \Rrightarrow e_2'$, then*

1. *If $e_1 \Rrightarrow e_2$ then $e_1[e_1'/x] \Rrightarrow e_2[e_2'/x]$, and*
2. *If $T_1 \Rrightarrow T_2$ then $T_1[e_1/x] \Rrightarrow T_2[e_2/x]$.*

Proof. By induction on $e_1 \Rrightarrow e_2$ and $T_1 \Rrightarrow T_2$. P_Refl and P_TRefl are immediate. P_ROpis by assumption. P_RRefl is by the IH. For P_Blame, the substitution has no effect. In every other case, the result is by the IH and reapplication of the rule.

Lemma 3 (Reduction ($\rightsquigarrow$) implies parallel reduction). *If $e_1 \rightsquigarrow e_2$, then $e_1 \Rrightarrow e_2$.*

Proof. Immediate: each rule for $\leadsto$ has a corresponding P_R… rule.

Lemma 4 (Reduction $\longrightarrow$ implies parallel reduction). *If $e_1 \longrightarrow e_2$, then $e_1 \Rrightarrow e_2$.*

Proof. By induction on the reduction, using the IH in the E_Compat case and Lemma 3 in the E_Reduce case. E_Blame is by P_Blame.

Lemma 5 (Parallel reduction implies cotermination). *If $e \Rrightarrow e'$ then*

1. *$e \longrightarrow^* v$ implies $e' \longrightarrow^* v'$ such that $v \Rrightarrow v'$.*
2. *$e' \longrightarrow^* v'$ implies $e \longrightarrow^* v$ such that $v \Rrightarrow v'$.*

Proof. This proof is standard, but finicky. The proof for a language with similar but slightly simpler semantics (Greenberg, Pierce, and Weirich [7] comprises 20 lemmas in 4000 lines of Coq; we expect to be able to adapt that proof.

Lemma 6 (Term substitution and type equivalence). *If $e \longrightarrow e'$ then $T[e/x] \equiv T[e'/x]$.*

Proof. Direct. By Lemma A4, $e \Rrightarrow e'$. By reflexivity, $T \Rrightarrow T$. By Lemma 2, $T[e/x] \Rrightarrow T[e'/x]$. We therefore conclude that $T[e/x] \equiv T[e'/x]$.

Lemma 7 (Type substitution and type equivalence). *If $T_1 \equiv T_2$ then $T_1[T/\alpha] \equiv T_2[T/\alpha]$.*

Proof. By induction on $T_1 \equiv T_2$.
(ParRed): If $T_1 \Rrightarrow T_2$, then we can see $T_1[T/\alpha] \Rrightarrow T_2[T/\alpha]$ by induction on the parallel reduction derivation, using reflexivity when we must replace $\alpha \Rrightarrow \alpha$ derivations with $T \Rrightarrow T$.
(Sym): By the IH.
(Trans): By the IHs.

Lemma 8 (Conversion and unref**).** *If $T_1 \equiv T_2$ then* $\text{unref}(T_1) \equiv \text{unref}(T_2)$.

Proof. By induction on the conversion relation.
(ParRed): By the IHs if using P_Refine; immediate, otherwise.
(Sym): By the IH and Sym.
(Trans): By the IHs and Trans.

Lemma 9 (Term weakening). *If x is fresh and $\Gamma \vdash T'$ then*

1. *$\Gamma, \Gamma' \vdash e : T$ implies $\Gamma, x{:}T', \Gamma \vdash e : T$,*
2. *$\Gamma, \Gamma' \vdash T$ implies $\Gamma, x{:}T', \Gamma' \vdash T$, and*
3. *$\vdash \Gamma, \Gamma'$ implies $\vdash \Gamma, x{:}T', \Gamma'$.*

Proof. By induction on e, T, and Γ'.
($\Gamma' = \emptyset$): $\Gamma \vdash T'$, so by WF_ExtendVar.
($\Gamma' = \Gamma'', y{:}T$): $\Gamma \vdash T'$, and $\vdash \Gamma, \Gamma'', y{:}T$. By the IH on $\vdash \Gamma, \Gamma''$, we have $\vdash \Gamma, x{:}T', \Gamma''$. By the IH on $\Gamma, \Gamma'' \vdash T$, we have $\Gamma, x{:}T', \Gamma'' \vdash T$. By WF_ExtendVar, $\vdash \Gamma, x{:}T', \Gamma'', y{:}T$.

$(\Gamma' = \Gamma'', \alpha)$: $\Gamma \vdash T'$ and $\vdash \Gamma, \Gamma'', \alpha$. By the IH on $\vdash \Gamma, \Gamma''$, we have $\vdash \Gamma, x{:}T', \Gamma''$. By the IH on $\Gamma, \Gamma'' \vdash T$, we have $\Gamma, x{:}T', \Gamma'' \vdash T$. By WF_ExtendTVar, $\vdash \Gamma, x{:}T', \Gamma'', \alpha$.

$(T = B)$: By WF_Base and the IH.

$(T = \alpha)$: By WF_TVar and the IH.

$(T = \{y{:}T \mid e\})$: $\Gamma, x{:}T', \Gamma'', y{:}T \vdash e : \mathsf{Bool}$ by the IH with $\Gamma' = \Gamma', y{:}T$, so by WF_Refine.

$(T = y{:}T_1 \to T_2)$: By the IH on $\Gamma, \Gamma' \vdash T_1$ and the IH on $\Gamma, \Gamma', y{:}T_1$ (with $\Gamma' = \Gamma', y{:}T_1$), then by WF_Fun.

$(T = \forall\alpha.\ T)$: By the IH (with $\Gamma' = \Gamma', \alpha$) on $\Gamma, \Gamma', \alpha \vdash T$, we have $\Gamma, x{:}T', \Gamma', \alpha \vdash T$. By WF_Forall.

(T_Conv,T_Exact,T_Forget): The argument is the same for all terms, so: since $\vdash \Gamma, x{:}T', \Gamma'$, we can reapply T_Conv, T_Exact, or T_Forget, respectively. In the rest of this proof, we won't bother considering these rules.

$(e = y)$: T_Var applied. $\vdash \Gamma, x{:}T', \Gamma'$ by the IH. If $y{:}T \in \Gamma, \Gamma'$, then $y{:}T \in \Gamma, x{:}T', \Gamma'$, so by T_Var.

$(e = k)$: T_Const applied: context well formedness is preserved by the IH, and type assignment ignores the context. So by T_Const.

$(e = \mathrm{op}\,(e_1, \ldots, e_n))$: T_Op applied. Each of the e_i can be typed in the weakened context by the IH, and the type of op ignores the context. So by T_Op.

$(e = \lambda y{:}T_1.\ e_{12})$: T_Abs applied. By the IH we find $\Gamma, x{:}T', \Gamma' \vdash T_1$. Also by the IH (with $\Gamma' = \Gamma', y{:}T_1$) we find $\Gamma, x{:}T', \Gamma', y{:}T_1 \vdash e_{12} : T_2$. So by T_Abs.

$(e = e_1\ e_2)$: T_App applied. By the IHs and reapplication of T_App.

$(e = \Lambda\alpha.\ v)$: T_TAbs applied. By the IH, with $\Gamma' = \Gamma', \alpha$, we find $\Gamma, x{:}T', \Gamma', \alpha \vdash v : T$, so by T_TAbs.

$(e = e_1\ T_2)$: T_TApp applied. By the IHs and reapplication of T_TApp.

$(e = \langle T_1 \Rightarrow T_2 \rangle^l)$: T_Cast applied. By the IHs and reapplication of T_Cast; compatibility ignores the context.

$(e = \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l)$: T_Check applied. Since $\vdash \Gamma, x{:}T', \Gamma'$, we can reapply T_Check.

$(e = \Uparrow l{:})$ T_Blame applied. Since $\vdash \Gamma, x{:}T', \Gamma'$, we can reapply T_Blame.

Lemma 10 (Type weakening). *If α is fresh then*

1. $\Gamma, \Gamma' \vdash e : T$ *implies* $\Gamma, \alpha, \Gamma \vdash e : T$,
2. $\Gamma, \Gamma' \vdash T$ *implies* $\Gamma, \alpha, \Gamma' \vdash T$, *and*
3. $\vdash \Gamma, \Gamma'$ *implies* $\vdash \Gamma, \alpha, \Gamma'$.

Proof. By induction on e, T, and Γ'.

$(\Gamma' = \emptyset)$: $\vdash \Gamma$, so we have $\vdash \Gamma, \alpha$ by WF_ExtendTVar.

$(\Gamma' = \Gamma'', y{:}T)$: $\Gamma \vdash T'$, and $\vdash \Gamma, \Gamma'', y{:}T$. By the IH on $\vdash \Gamma, \Gamma''$, we have $\vdash \Gamma, \alpha, \Gamma''$. By the IH on $\Gamma, \Gamma'' \vdash T$, we have $\Gamma, \alpha, \Gamma'' \vdash T$. By WF_TVar, $\vdash \Gamma, \alpha, \Gamma'', y{:}T$.

$(\Gamma' = \Gamma'', \alpha')$: $\Gamma \vdash T'$ and $\vdash \Gamma, \Gamma'', \alpha'$. By the IH on $\vdash \Gamma, \Gamma''$, we have $\vdash \Gamma, \alpha, \Gamma''$. By WF_ExtendTVar, $\vdash \Gamma, \alpha, \Gamma'', \alpha'$.

$(T = B)$: By WF_Base and the IH.

($T = \alpha'$): By WF_TVar and the IH.

($T = y{:}T_1 \to T_2$): By the IH on $\Gamma, \Gamma' \ \vdash \ T_1$ and the IH on $\Gamma, \Gamma', y{:}T_1$ (with $\Gamma' = \Gamma', y{:}T_1$), then by WF_Fun.

($T = \forall\alpha'.\ T$): By the IH (with $\Gamma' = \Gamma', \alpha'$) on $\Gamma, \Gamma', \alpha' \ \vdash \ T$, we have $\Gamma, \alpha, \Gamma', \alpha' \ \vdash \ T$. By WF_Forall.

($T = \{y{:}T \mid e\}$): $\Gamma, \alpha, \Gamma'', y{:}T \vdash e : \mathsf{Bool}$ by the IH with $\Gamma' = \Gamma', y{:}T$, so by WF_Refine.

(T_Conv,T_Exact,T_Forget): The argument is the same for all terms, so: since $\vdash \Gamma, \alpha, \Gamma'$, we can reapply T_Conv, T_Exact, or T_Forget, respectively.

($e = y$): T_Var applied. $\vdash \Gamma, \alpha, \Gamma'$ by the IH. If $y{:}T \in \Gamma, \Gamma'$, then $y{:}T \in \Gamma, \alpha, \Gamma'$, so by T_Var.

($e = k$): T_Const applied: context well formedness is preserved by the IH, and type assignment ignores the context. So by T_Const.

($e = \mathrm{op}\,(e_1, \ldots, e_n)$): T_Op applied. Each of the ei can be typed in the weakened context by the IH, and the type of op ignores the context. So by T_Op.

($e = \lambda y{:}T_1.\ e_{12}$): T_Abs applied. By the IH we find $\Gamma, \alpha, \Gamma' \ \vdash \ T_1$. Also by the IH (with $\Gamma' = \Gamma', y{:}T_1$) we find $\Gamma, \alpha, \Gamma', y{:}T_1 \vdash e_{12} : T_2$. So by T_Abs.

($e = e_1\ e_2$): T_App applied. By the IHs and reapplication of T_App.

($e = \Lambda\alpha'.\ v$): T_TAbs applied. By the IH, with $\Gamma' = \Gamma', \alpha'$, we find $\Gamma, \alpha, \Gamma', \alpha' \vdash v : T$, so by T_TAbs.

($e = e_1\ T_2$): T_TApp applied. By the IHs and reapplication of T_TApp.

($e = \langle T_1 \Rightarrow T_2\rangle^l$): T_Cast applied. By the IHs and reapplication of T_Cast; compatibility ignores the context.

($e = \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$): T_Check applied. Since $\vdash \Gamma, \alpha, \Gamma'$, we can reapply T_Check.

($e = \Uparrow l$:) T_Blame applied. Since $\vdash \Gamma, \alpha, \Gamma'$, we can reapply T_Blame.

Lemma 11 (Term substitution). *If $\Gamma \vdash e' : T'$, then*

1. *if $\Gamma, x{:}T', \Gamma' \vdash e : T$ then $\Gamma, \Gamma'[e'/x] \vdash e[e'/x] : T[e'/x]$,*
2. *if $\Gamma, x{:}T', \Gamma' \ \vdash \ T$ then $\Gamma, \Gamma'[e'/x] \ \vdash \ T[e'/x]$, and*
3. *if $\vdash \Gamma, x{:}T', \Gamma'$ then $\vdash \Gamma, \Gamma'[e'/x]$.*

Proof. By induction on e, T, and Γ'. In the first two clauses, we are careful to leave Γ' general.

($\Gamma' = \emptyset$): We already have $\vdash \Gamma$ by assumption..

($\Gamma' = \Gamma'', y{:}T$): By the IH, $\vdash \Gamma, \Gamma''[e'/x]$ and $\Gamma, \Gamma''[e'/x] \ \vdash \ T[e'/x]$. By WF_ExtendVar, $\vdash \Gamma, \Gamma''[e'/x], y{:}T[e'/x]$.

($\Gamma' = \Gamma'', \alpha$): By the IH, $\vdash \Gamma, \Gamma''[e'/x], \alpha$. By WF_ExtendTVar.

($T = B$): By the IH and WF_Base.

($T = \alpha$): By the IH and WF_TVar.

($T = y{:}T_1 \to T_2$): By the IHs we have $\Gamma, \Gamma'[e'/x] \ \vdash \ T_1[e'/x]$ and $\Gamma, \Gamma'[e'/x], y{:}T_1[e'/x] \ \vdash \ T_2[e'/x]$. By WF_Fun, we find $\Gamma, \Gamma'[e'/x] \ \vdash \ y{:}T_1[e'/x] \to T_2[e'/x]$.

($T = \forall\alpha.\ T$): By the IH we find $\Gamma, \Gamma'[e'/x], \alpha \ \vdash \ T[e'/x]$. So by WF_Forall.

($T = \{y{:}T \mid e\}$): By the IH we have $\Gamma, \Gamma'[e'/x], y{:}T \vdash e[e'/x] : \mathsf{Bool}$. By WF_Refine, we find $\Gamma, \Gamma'[e'/x] \ \vdash \ \{y{:}T \mid e[e'/x]\}$.

(T_Conv,T_Exact,T_Forget): This reasoning is the same for all terms: context well formedness is by the IH. We reapply the original rule, noting that the actual term and its type are both closed, so the substitution has no effect.
($e = y$): T_Var applied. Context well formedness is by the IH.

If $y = x$, then by weakening we can take $\Gamma \vdash x : T$ to $\Gamma, \Gamma' \vdash x : T$. In this case, x doesn't occur in T, so $T[e'/x] = T$, and we are done by T_Var.

If $y \neq x$, there are two cases.

- If $y{:}T \in \Gamma'$, then we have $y{:}T[e'/x] \in \Gamma[e'/x]$, which yields the correct result with T_Var.
- If $y{:}T \in \Gamma$, then x doesn't occur in Γ, and we can derive $y{:}T[e'/x] \in \Gamma, \Gamma'[e'/x]$ (which is the same as $y{:}T \in \Gamma, \Gamma'[e'/x]$) with T_Var.

($e = k$): T_Const applied. Context well formedness is by the IH, and $\mathrm{ty}\,(k)$ is unaffected. By T_Const.
($e = \mathrm{op}\,(e_1, \ldots, e_n)$): T_Op applied. We find substituted types for each of the e_i by the IH, noting that x doesn't occur in $\mathrm{ty}\,(\,\mathrm{op}\,)$ at all. By T_Op.
($e = \lambda y{:}T_1.\ e_{12}$): T_Abs applied. By the IH, $\Gamma, \Gamma'[e'/x] \ \vdash \ T_1[e'/x]$. Using $\Gamma', y{:}T_1$ as Γ', the IH also gives us $\Gamma, \Gamma'[e'/x], y{:}T_1[e'/x] \vdash e_{12}[e'/x] : T_2[e'/x]$. By T_Abs, we find $\Gamma, \Gamma'[e'/x] \vdash \lambda y{:}T_1[e'/x].\ e_{12}[e'/x] : (y{:}T_1[e'/x] \to T_2[e'/x])$.
($e = e_1\ e_2$): By the IHs and reapplication of T_App.
($e = \Lambda\alpha.\ v$): T_TAbs applied. By the IH, $\Gamma, \Gamma'[e'/x], \alpha \vdash v[e'/x] : T[e'/x]$. By T_TAbs, $\Gamma, \Gamma'[e'/x] \vdash (\Lambda\alpha.\ v)[e'/x] : \forall\alpha.\ T[e'/x]$.
($e = e_1\ T_2$): By the IHs and repplication of T_TApp.
($e = \langle T_1 \Rightarrow T_2 \rangle^l$): By the IHs and reapplication of T_Cast, noting that compatibility is unaffected.
($e = \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l$): T_Check applied. Since $\vdash \Gamma, \Gamma'[e'/x]$ by the IH, we can reapply T_Check.
($e = \Uparrow l$): Context well formedness is by the IH; by T_Blame.

Lemma 12 (Type substitution preserves compatibility). *If $T_1 \parallel T_2$ then $T_1[T'/\alpha] \parallel T_2[T'/\alpha]$.*

Proof.

Lemma 13 (Type substitution manipulation). *If α' does not occur in T', then $(T_1[T'/\alpha])[T_2[T'/\alpha]/\alpha'] = (T_1[T_2/\alpha'])[T'/\alpha]$.*

Proof.

Lemma 14 (Type substitution). *If $\Gamma \ \vdash \ T'$ then*

1. *if $\Gamma, \alpha, \Gamma' \vdash e : T$, then $\Gamma, \Gamma'[T'/\alpha] \vdash e[T'/\alpha] : T[T'/\alpha]$,*
2. *if $\Gamma, \alpha, \Gamma' \ \vdash \ T$, then $\Gamma, \Gamma'[T'/\alpha] \ \vdash \ T[T'/\alpha]$, and*
3. *if $\vdash \Gamma, \alpha, \Gamma'$, then $\vdash \Gamma, \Gamma'[T'/\alpha]$.*

Proof. By induction on e, T, and Γ'.
($\Gamma' = \emptyset$): We already have $\vdash \Gamma$ by assumption.

($\Gamma' = \Gamma'', y{:}T$): By the IH, $\vdash \Gamma, \Gamma''[T'/\alpha]$ and $\Gamma, \Gamma''[T'/\alpha] \vdash T[T'/\alpha]$. By WF_ExtendVar, $\vdash \Gamma, \Gamma''[T'/\alpha], y{:}T[T'/\alpha]$.
($\Gamma' = \Gamma'', \alpha'$): By the IH, $\vdash \Gamma, \Gamma''[T'/\alpha], \alpha'$. By WF_ExtendTVar.
($T = B$): By the IH and WF_Base.
($T = \alpha'$): There are two cases.

If $\alpha = \alpha'$, then, since $\vdash \Gamma, \Gamma'[T'/\alpha]$ (by the IH), we can weaken $\Gamma \vdash T$ to see $\Gamma, \Gamma' \vdash T'$.

If $\alpha \neq \alpha'$, then $\Gamma, \Gamma'[T'/\alpha] \vdash \alpha'$ because α' is unaffected by the substitution whether or not α' is in Γ or Γ'.
($T = y{:}T_1 \to T_2$): By the IHs we have $\Gamma, \Gamma'[T'/\alpha] \vdash T_1[T'/\alpha]$ and $\Gamma, \Gamma'[T'/\alpha], y{:}T_1[T'/\alpha] \vdash T_2[T'/\alpha]$. By WF_Fun, we find $\Gamma, \Gamma'[T'/\alpha] \vdash y{:}T_1[T'/\alpha] \to T_2[T'/\alpha]$.
($T = \forall\alpha'.\ T$): By the IH we find $\Gamma, \Gamma'[T'/\alpha], \alpha' \vdash T[T'/\alpha]$. So by WF_Forall.
($T = \{x{:}T \mid e\}$): By the IH we have $\Gamma, \Gamma'[T'/\alpha], y{:}T[T'/\alpha] \vdash e[T'/\alpha] : \mathsf{Bool}$. By WF_Refine, we find $\Gamma, \Gamma'[T'/\alpha] \vdash \{y{:}T[T'/\alpha] \mid e[T'/\alpha]\}$.
(T_Conv,T_Exact,T_Forget): This is the same for all terms: context well formedness is by the IH; by the original rule, noting that the actual term and its type are both closed, so the substitution has no effect.
($e = x$): T_Var applied. Context well formedness is by the IH. If $x{:}T \in \Gamma$, then $x{:}T \in \Gamma, \Gamma'$. If $x{:}T \in \Gamma'$, then $x{:}T[T'/\alpha] \in \Gamma'[T'/\alpha]$. By T_Var.
($e = k$): T_Const applied. Context well formedness is by the IH, and ty is unaffected. By T_Const.
($e = \mathrm{op}\,(e_1, \ldots, e_n)$): T_Op applied. We find substituted types for each of the ei by the IH, noting that x doesn't occur in $\mathrm{ty}\,(\,\mathrm{op}\,)$ at all. By T_Op.
($e = \lambda y{:}T_1.\ e_{12}$): T_Abs applied. By the IH, $\Gamma, \Gamma'[T'/\alpha] \vdash T_1[T'/\alpha]$. Using $\Gamma', x{:}T_1$ as Γ', the IH also gives us $\Gamma, \Gamma'[T'/\alpha], x{:}T_1[T'/\alpha] \vdash e_{12}[T'/\alpha] : T_2[T'/\alpha]$. By T_Abs, we find $\Gamma, \Gamma'[T'/\alpha] \vdash \lambda x{:}T_1[T'/\alpha].\ e_{12}[T'/\alpha] : (x{:}T_1[T'/\alpha] \to T_2[T'/\alpha])$.
($e = e_1\ e_2$): By the IHs and reapplication T_App.
($e = \Lambda\alpha'.\ v$): By the IH on the T_TAbs derivation, $\Gamma, \Gamma'[T'/\alpha], \alpha' \vdash v[T'/\alpha] : T[T'/\alpha]$. By T_TAbs, $\Gamma, \Gamma'[T'/\alpha] \vdash (\Lambda\alpha'.\ v)[T'/\alpha] : \forall\alpha'.\ T[T'/\alpha]$.
($e = e_1\ T_2$): By the IHs and reapplication of T_TApp.
($e = \langle T_1 \Rightarrow T_2\rangle^l$): By the IHs and reapplication T_Cast, noting that compatibility is unaffected.
($e = \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$): T_Check applied. Since $\vdash \Gamma, \Gamma'[T'/\alpha]$ by the IH, we can reapply T_Check.
($e = \Uparrow l$): Context well formedness is by the IH; by reapplication of T_Blame.

Lemma 15 (Conversion of unrefined types). *If* $T_1 \equiv T_2$ *then* $\mathrm{unref}(T_1) \equiv \mathrm{unref}(T_2)$.

Proof. By induction on the derivation of $\equiv$.
(P_Refl): Immediate, since unref does nothing.
(P_Refine): By the IH and one level of refinement stripping.
(P_Fun): Immediate, since unref does nothing.
(P_Forall): Immediate, since unref does nothing.

(Sym): By the IH and Sym.
(Trans): By the IH and Trans.

Lemma 16 (Lambda inversion). *If $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$, then*

1. $\Gamma \ \vdash \ T_1$,
2. $\Gamma, x{:}T_1 \vdash e_{12} : T_2$, *and*
3. $x{:}T_1 \to T_2 \equiv \text{unref}(T)$.

Proof. By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.
(T_Abs): $\text{unref}(T) = T = x{:}T_1 \to T_2$, and the derivation is by inversion. Conversion is by reflexivity.
(Conv): We have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$; by inversion, $T \equiv T'$ and $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T'$. By the IH on this second derivation, we find the derivations we desire (in the empty context—we must use weakening to get them in Γ) and the fact that $x{:}T_1 \to T_2 \equiv \text{unref}(T')$. Since we have $T' \equiv T$, we have $x{:}T_1 \to T_2 \equiv \text{unref}(T') \equiv \text{unref}(T)$ by transitivity.
(Exact): $T = \{x{:}T' \mid e\}$, and we have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : \{x{:}T' \mid e\}$; by inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T'$. By the IH, we find the derivations we desire (up to weakening), and $x{:}T_1 \to T_2 \equiv \text{unref}(T')$. Since $\text{unref}(T') = \text{unref}(\{x{:}T' \mid e\})$, we have the conversion immediately, as well.
(Forget): We have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$; by inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : \{x{:}T \mid e\}$. By the IH on this latter derivation, we find the derivations we desire (up to weakening) and the conversion $x{:}T_1 \to T_2 \equiv \text{unref}(\{x{:}T \mid e\})$. Since $\text{unref}(\{x{:}T \mid e\}) = \text{unref}(T)$, we also have the conversion.

Lemma 17 (Cast inversion). *If $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l : T$, then*

1. $\Gamma \ \vdash \ T_1$,
2. $\Gamma \ \vdash \ T_2$,
3. $T_1 \parallel T_2$, *and*
4. $\_{:}T_1 \to T_2 \equiv \text{unref}(T)$ *(i.e., T_2 does not mention the dependent variable).*

Proof. By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.
(T_Cast): $T = \_{:}T_1 \to T_2$, and the derivation is by inversion. Conversion is by reflexivity. T_2 does not mention the variable.
(Conv): $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l : T$; by inversion, $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : T'$ and $T' \equiv T$. By the IH, we have $\emptyset \ \vdash \ T_1$ and $\emptyset \ \vdash \ T_2$ (which weaken to the derivations we want), as well as $T_1 \parallel T_2$ and $\_{:}T_1 \to T_2 \equiv \text{unref}(T')$. But $\text{unref}(T') \equiv \text{unref}(T)$, so we have the conversion we want by transitivity.
(Exact): $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l : \{x{:}T \mid e\}$; by inversion, $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : T$. By the IH, we have $\emptyset \ \vdash \ T_1$ and $\emptyset \ \vdash \ T_2$ (which weaken to the derivations we want). We also have $T_1 \parallel T_2$ and $\_{:}T_1 \to T_2 \equiv \text{unref}(T)$ — which is equal to $\text{unref}(\{x{:}T \mid e\})$, so we're done.

(Forget): $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l : T$; by inversion, $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : \{x{:}T \mid e\}$. By the IH, $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$, so we find the derivations we want by weakening. We also have $T_1 \parallel T_2$ and $\_{:}T_1 \rightarrow T_2 \equiv \text{unref}(T)$, which is equal $\text{unref}(\{x{:}T \mid e\})$, so this case is complete.

Lemma 18 (Application inversion). *If $\Gamma \vdash e_1\, e_2 : T$, then*

1. $\Gamma \vdash e_1 : (x{:}T_1 \rightarrow T_2)$,
2. $\Gamma \vdash e_2 : T_1$, *and*
3. $T_2[e_2/x] \equiv T$.

Proof. By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.
(T_App): We find the typings we want by inversion. Since $T = T_2[e_2/x]$, conversion is by reflexivity.
(Conv): By the IH on $\emptyset \vdash e_1\, e_2 : T'$, we find the typings we want (up to weakening), and $T_2[e_2/x] \equiv T'$. By inversion, $T' \equiv T$, so by transitivity we have $T_2[e_2/x]$.
(Exact): Only applies to values, and $e_1\, e_2$ cannot be a value.
(Forget): Only applies to values, and $e_1\, e_2$ cannot be a value.

Lemma 19 (Big lambda inversion). *If $\Gamma \vdash \Lambda\alpha.\ e : T$, then*

1. $\Gamma, \alpha \vdash e : T'$ *and*
2. $\forall\alpha.\ T' \equiv \text{unref}(T)$.

Proof. By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.
(T_TAbs): $T = \forall\alpha.\ T'$. Conversion is by reflexivity.
(T_Conv): By the IH on $\emptyset \vdash \Lambda\alpha.\ e : T''$, using weakening to recover the typing derivation and using transitivity of conversion through unref to find $\text{unref}(T'') \equiv \text{unref}(T)$.
(T_Exact): $T = \{x{:}T_0 \mid e_0\}$ for some T_0 and e_0. By the IH on $\emptyset \vdash \Lambda\alpha.\ e : T_0$, using weakening for the derivation and the fact that $\text{unref}(\{x{:}T_0 \mid e_0\}) = \text{unref}(T_0)$ to find conversion.
(T_Forget): By the IH on $\emptyset \vdash \Lambda\alpha.\ e : \{x{:}T \mid e_0\}$, using weakening for the derivation and the fact that $\text{unref}(\{x{:}T \mid e_0\}) = \text{unref}(T)$ to find conversion.

Lemma 20 (Canonical forms). *If $\emptyset \vdash v : T$, then:*

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

Proof. By induction on the typing derivation.
(T_Var): Contradictory: variables are not values.

(T_Const): $\emptyset \vdash k : T$ and $\mathrm{unref}(T) = B$; we are in case 1. By assumption, $k \in \mathcal{K}_B$.
(T_Op): Contradictory: $\mathrm{op}\,(e_1, \ldots, e_n)$ is not a value.
(T_Abs): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T$ and $T = \mathrm{unref}(T) = x{:}T_1 \to T_2$; we are in case 2a. Conversion is by reflexivity.
(T_App): Contradictory: $e_1\, e_2$ is not a value.
(T_TAbs): $\emptyset \vdash \Lambda\alpha.\ v' : \forall\alpha.\ T$; we are in case 3. It is immediate that $v = \Lambda\alpha.\ v'$, and conversion is by reflexivity.
(T_TApp): Contradictory: $e\ T$ is not a value.
(T_Cast): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : \_{:}T_1 \to T_2$; we are in case 2b. It is immediate that $v = \langle T_1 \Rightarrow T_2\rangle^l$. Conversion is by reflexivity.
(T_Check): Contradictory: $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$ is not a value.
(T_Blame): Contradictory: $\Uparrow l$ is not a value.
(T_Conv): $\emptyset \vdash v : T$; by inversion, $\emptyset \vdash v : T'$ and $T' \equiv T$. We find an appropriate form for $\mathrm{unref}(T')$ by the IH on $\emptyset \vdash v : T'$. We go by cases, in each case reproving whatever case was found in the IH and finding conversions by transitivity and congruence:

- Case 1: $\mathrm{unref}(T') = B$ and $v = k \in \mathcal{K}_B$. Since $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$, we know that $\mathrm{unref}(T) = B$, which is all we needed to show.
- Case 2a: $\mathrm{unref}(T') = x{:}T_1' \to T_2'$ and $v = \lambda x{:}T_1''.\ e_{12}$ and $T_1'' \equiv T_1'$. Since $T' \equiv T$, we have $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ and $T_1' \equiv T_1$; by transitivity, we have $\mathrm{unref}(T) \equiv x{:}T_1 \to T_2$. Since $T_1'' \equiv T_1'$, we have $T_1'' \equiv T_1$, and we have recovered the conversions we need.
- Case 2b: $\mathrm{unref}(T') = x{:}T_1' \to T_2'$ and $v = \langle T_1' \Rightarrow T_2'\rangle^l$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$. Since $T' \equiv T$, we have $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$; by transitivity, we have $\mathrm{unref}(T) \equiv x{:}T_1 \to T_2$. Since $T_1'' \equiv T_1'$, we have $T_1'' \equiv T_1$. We similarly find $T_2'' \equiv T_2$, and we have recovered the conversions we need.
- Case 3: $\mathrm{unref}(T') = \forall\alpha.\ T$ and v is $\Lambda\alpha.\ v'$. Since $T' \equiv T$, then $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$.

(T_Exact): $\emptyset \vdash v : \{x{:}T \mid e\}$; by inversion, $\emptyset \vdash v : T$. Noting that $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$, we apply the IH. Unlike the previous case, we need not change the conversion—it's in terms of the unrefined type.
(T_Forget): $\emptyset \vdash v : T$; by inversion $\emptyset \vdash v : \{x{:}T \mid e\}$. By the IH (noting $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$), so we use the IH's conversion directly.

Lemma 21 (Context and type well formedness).

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

Proof. By induction on the typing and well formedness derivations.
(WF_Base): $\Gamma \ \vdash\ B$. We find $\vdash \Gamma$ by inversion.
(WF_TVar): $\Gamma \ \vdash\ \alpha$. We find $\vdash \Gamma$ by inversion.

(WF_Fun): $\Gamma \ \vdash \ x{:}T_1 \to T_2$. By inversion, $\Gamma \ \vdash \ T_1$ and $\Gamma, x{:}T_1 \ \vdash \ T_2$. By the IH on $\Gamma \ \vdash \ T_1$, we find $\vdash \Gamma$.

(WF_Forall): $\Gamma \ \vdash \ \forall\alpha.\ T$; by inversion, $\Gamma, \alpha \ \vdash \ T$. By the IH on $\Gamma, \alpha \ \vdash \ T$, we have $\vdash \Gamma, \alpha$. By further inversion, we have that $\vdash \Gamma$.

(WF_Refine): $\Gamma \ \vdash \ \{x{:}T \mid e\}$; by inversion, $\Gamma \ \vdash \ T$ and $\Gamma, x{:}T \vdash e : \mathsf{Bool}$. By the IH on $\Gamma \ \vdash \ T$, we have $\vdash \Gamma$.

(T_Var): $\Gamma \vdash x : T$; by inversion, $\vdash \Gamma$ and $x{:}T \in \Gamma$. Since $x{:}T \in \Gamma$, we know that $\Gamma = \Gamma_1, x{:}T, \Gamma_2$. We can open up the derivation of Γ's well formedness to find $\Gamma_1 \ \vdash \ T$. By weakening (Lemmas 9 and 10), we find $\Gamma \ \vdash \ T$.

(T_Const): $\Gamma \vdash k : \mathrm{ty}\,(k)$; by inversion, $\vdash \Gamma$. We assume that ty produces well formed types.

(T_Op): $\Gamma \vdash \mathrm{op}\,(e_1, \ldots, e_n) : \sigma(T)$; where $\mathrm{ty}\,(\,\mathrm{op}\,) = x_1 : T_1 \to \ldots \to x_n : T_n \to T$. By inversion, $\vdash \Gamma$ and $\Gamma \vdash e_i : \sigma(T_i)$. We assume that ty produces well formed types; the substituted type is well formed by term substitution (Lemma 11).

(T_Abs): $\Gamma \vdash \lambda x{:}T_1.\ e_2 : (x{:}T_1 \to T_2)$; by inversion, $\Gamma \ \vdash \ T_1$ and $\Gamma, x{:}T_1 \vdash e_2 : T_2$. By the IH on the first derivation, we have $\vdash \Gamma$. By the IH on the second derivation, we have $\Gamma, x{:}T_1 \ \vdash \ T_2$. By WF_Fun, we can see $\Gamma \ \vdash \ x{:}T_1 \to T_2$.

(T_App): $\Gamma \vdash e_1\, e_2 : T_2[e_2/x]$; by inversion, $\Gamma \vdash e_1 : (x{:}T_1 \to T_2)$ and $\Gamma \vdash e_2 : T_1$. By the IH on the first derivation, $\vdash \Gamma$ and $\Gamma \ \vdash \ x{:}T_1 \to T_2$. By inversion of that typing derivation, $\Gamma, x{:}T_1 \ \vdash \ T_2$. By substitution (Lemma 11) on $\Gamma, x{:}T_1 \ \vdash \ T_2$ and $\Gamma \vdash e_2 : T_1$, we have $\Gamma \ \vdash \ T_2[e_2/x]$.

(T_TAbs): $\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ T$; by inversion, $\Gamma, \alpha \vdash e : T$. By the IH, we have $\vdash \Gamma, \alpha$ and $\Gamma, \alpha \ \vdash \ T$. We can invert the context well formedness to see $\vdash \Gamma$. By WF_Forall, we can derive $\Gamma \ \vdash \ \forall\alpha.\ T$.

(T_TApp): $\Gamma \vdash e_1\ T_2 : T_1[T_2/\alpha]$; by inversion, we have $\Gamma \vdash e_1 : \forall\alpha.\ T_1$ and $\Gamma \ \vdash \ T_2$. By the IH on the first derivation, we have $\vdash \Gamma$ and $\Gamma \ \vdash \ \forall\alpha.\ T_1$. By inversion on this last, $\Gamma, \alpha \ \vdash \ T_1$. By type substitution (Lemma 14) on this derivation, $\Gamma \ \vdash \ T_1[T_2/\alpha]$.

(T_Cast): $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l : ({\_}{:}T_1 \to T_2)$; by inversion, $\Gamma \ \vdash \ T_1$ and $\Gamma \ \vdash \ T_2$ (and, less relevantly, $T_1 \parallel T_2$). By the IH on either derivation, we have $\vdash \Gamma$. Let x be fresh for Γ, T_1, and T_2—then, by weakening we have $\Gamma, x{:}T_1 \ \vdash \ T_2$. By WF_Fun, we can now derive $\Gamma \ \vdash \ x{:}T_1 \to T_2$. Since x is fresh, this means that $\Gamma \ \vdash \ T_1 \to T_2$.

(T_Check): $\Gamma \vdash \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l : \{x{:}T \mid e_1\}$; by inversion, we have, among other things, $\vdash \Gamma$ and $\emptyset \ \vdash \ \{x{:}T \mid e_1\}$. By repeated application of weakening (Lemmas 9 and 10), we have $\Gamma \ \vdash \ \{x{:}T \mid e_1\}$.

(T_Blame): $\Gamma \vdash \Uparrow l : T$; by inversion, $\vdash \Gamma$ and $\emptyset \ \vdash \ T$. By repeated application of weakening (Lemmas 9 and 10), we have $\Gamma \ \vdash \ T$.

(Conv): $\Gamma \vdash e : T'$; by inversion we can see $\vdash \Gamma$ and $\emptyset \ \vdash \ T'$. By repeated application of weakening (Lemmas 9 and 10), we have $\Gamma \ \vdash \ T'$.

(Exact): $\Gamma \vdash v : T$; by inversion, $\vdash \Gamma$ and $\emptyset \ \vdash \ \{x{:}T \mid e\}$. By repeated application of weakening (Lemmas 9 and 10), we have $\Gamma \ \vdash \ \{x{:}T \mid e\}$.

(Forget): $\Gamma \vdash v : T$; by inversion, $\vdash \Gamma$ and $\emptyset \vdash v : \{x{:}T \mid e\}$. By the IH on the second derivation, $\emptyset \ \vdash \ \{x{:}T \mid e\}$. By inversion, $\emptyset \ \vdash \ T$. By repeated application of weakening (Lemmas 9 and 10), we have $\Gamma \ \vdash \ T$.

Theorem 22 (Progress). *If $\emptyset \vdash e : T$, then either*

1. $e \longrightarrow e'$, *or*
2. $e = r$.

Proof. By induction on the typing derivation.
(T_Var): Contradictory: there is no derivation $\emptyset \vdash x : T$.
(T_Const): $\emptyset \vdash k : \mathrm{ty}\,(k)$. In this case, $e = k$ is a result.
(T_Op): $\emptyset \vdash \mathrm{op}\,(e_1, \ldots, e_n) : \sigma(T)$, where $\mathrm{ty}\,(\,\mathrm{op}\,) = x_1 : T_1 \to \ldots \to x_n : T_n \to T$. By inversion, $\emptyset \vdash e_i : \sigma(T_i)$. Applying the IH from left to right, each of the e_i either steps or is a result.

Suppose everything to the left of e_i is a value. Then either e_i steps or is a result. If $e_i \longrightarrow e_i'$, then $\mathrm{op}\,(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow \mathrm{op}(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)$ by E_Compat. One the other hand, if e_i is a result, there are two cases. If $e_i = \Uparrow l$, then the original expression steps to $\Uparrow l$ by E_Blame. If e_i is a value, we can continue this process for each of the operation's arguments. Eventually, all of the operations arguments are values. We assume that if $\mathrm{op}\,(v_1, \ldots, v_n)$ is well typed, then E_Op applies.
(T_Abs): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \to T_2)$. In this case, $e = \lambda x{:}T_1.\ e_{12}$ is a result.
(T_App): $\emptyset \vdash e_1\, e_2 : T_2[e_2/x]$; by inversion, $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$ and $\emptyset \vdash e_2 : T_1$.

By the IH on the first derivation, e_1 steps or is a result. If e_1 steps, then the entire term steps by E_Compat. In the latter case, if e_1 is blame, we step by E_Blame. So e_1 is a value, v_1.

By the IH on the second derivation, e_2 steps or is a result. If e_2 steps, then by E_Compat. Otherwise, if e_2 is blame, we step by E_Blame. So e_2 is a value, v_2.

By canonical forms (Lemma 20) on $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$, there are two cases:

($e_1 = \lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$): In this case, $(\lambda x{:}T_1'.\ e_{12})\, v_2 \longrightarrow e_{12}[v_2/x]$ by E_Beta.
($e_1 = \langle T_1' \Rightarrow T_2' \rangle^l$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$): By cast inversion (Lemma 17), we know that $T_1' \parallel T_2'$. We determine which step is taken by cases on T_1' and T_2'.
($T_1' = B$):
($T_2' = B'$): It must be the case that $B = B'$, since $B \parallel B'$. By E_Refl, $\langle B \Rightarrow B \rangle^l\, v_2 \longrightarrow v_2$.
($T_2' = \alpha$ or $x{:}T_{21} \to T_{22}$ or $\forall \alpha.\ T_{22}$): Incompatible; contradictory.
($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = B$, then by E_Check, $\langle B \Rightarrow \{x{:}B \mid e\} \rangle^l\, v_2 \longrightarrow \langle \{x{:}B \mid e\}, e[v_2/x], v_2 \rangle^l$. Otherwise, by E_PreCheck, we have $\langle B \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l\, (\langle B \Rightarrow T_2'' \rangle^l\, v_2)$.
($T_1' = \alpha$):
($T_2' = \alpha'$): It must be the case that $\alpha = \alpha'$, since $\alpha \parallel \alpha'$. By E_Refl, $\langle \alpha \Rightarrow \alpha \rangle^l\, v_2 \longrightarrow v_2$.
($T_2' = B$ or $x{:}T_{21} \to T_{22}$ or $\forall \alpha.\ T_{22}$): Incompatible; contradictory.
($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = \alpha$, then by E_Check, $\langle \alpha \Rightarrow \{x{:}\alpha \mid e\} \rangle^l\, v_2 \longrightarrow \langle \{x{:}\alpha \mid e\}, e[v_2/x], v_2 \rangle^l$. Otherwise, $\langle \alpha \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l\, (\langle \alpha \Rightarrow T_2'' \rangle^l\, v_2)$ by E_PreCheck.

$\underline{(T'_1 = x{:}T_{11} \to T_{12})}$:

$\underline{(T'_2 = B \text{ or } \alpha \text{ or } \forall\alpha.\ T_{22})}$: Incompatible; contradictory.

$\underline{(T'_2 = x{:}T_{21} \to T_{22})}$: If $T'_1 = T'_2$, then $\langle T'_1 \Rightarrow T'_1\rangle^l\, v_2 \longrightarrow v_2$ by E_Refl. If not, then $\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v_2 \longrightarrow \lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)))$ by E_Fun.

$\underline{(T'_2 = \{x{:}T''_2 \mid e\})}$: If $T'_1 = T''_2$, then $\langle T'_1 \Rightarrow \{x{:}T'_1 \mid e\}\rangle^l\, v_2 \longrightarrow \langle\{x{:}T'_1 \mid e\}, e[v_2/x], v_2\rangle^l$ by E_Check. If not, then $\langle T'_1 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l\, v_2 \longrightarrow \langle T''_2 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l\, (\langle T'_1 \Rightarrow T''_2\rangle^l\, v_2)$ by E_PreCheck.

$\underline{(T'_1 = \forall\alpha.\ T_{12})}$:

$\underline{(T'_2 = B \text{ or } \alpha \text{ or } x{:}T_{21} \to T_{22})}$: Incompatible; contradictory.

$\underline{(T'_2 = \forall\alpha.\ T_{22})}$: If $T'_1 = T'_2$, then $\langle T'_1 \Rightarrow T'_2\rangle^l\, v_2 \longrightarrow v_2$ by E_Refl. If not, then $\langle \forall\alpha.\ T_{11} \Rightarrow \forall\alpha.\ T_{22}\rangle^l\, v_2 \longrightarrow \Lambda\alpha.\ (\langle T_{11} \Rightarrow T_{22}\rangle^l\, (v_2\, \alpha))$ by E_Forall.

$\underline{(T'_2 = \{x{:}T''_2 \mid e\})}$: If $T'_1 = T''_2$, then $\langle T'_1 \Rightarrow \{x{:}T'_1 \mid e\}\rangle^l\, v_2 \longrightarrow \langle\{x{:}T'_1 \mid e\}, e[v_2/x], v_2\rangle^l$ by E_Check. If not, then $\langle T'_1 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l\, v_2 \longrightarrow \langle T''_2 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l\, (\langle T'_1 \Rightarrow T''_2\rangle^l\, v_2)$ by E_PreCheck.

$\underline{(T'_1 = \{x{:}T''_1 \mid e'_1\})}$:

$\underline{(T'_2 = B \text{ or } \alpha \text{ or } x{:}T_{21} \to T_{22} \text{ or } \forall\alpha.\ T_{22})}$: We see $\langle\{x{:}T''_1 \mid e'_1\} \Rightarrow T'_2\rangle^l\, v_2 \longrightarrow \langle T''_1 \Rightarrow T'_2\rangle^l\, v_2$ by E_Forget.

$\underline{(T'_2 = \{x{:}T''_2 \mid e'_2\})}$: If $T'_1 = T'_2$, then $\langle T'_1 \Rightarrow T'_2\rangle^l\, v_2 \longrightarrow v_2$ by E_Refl. If $T'_1 = T''_2$, then $\langle T'_1 \Rightarrow \{x{:}T'_1 \mid e'_2\}\rangle^l\, v_2 \longrightarrow \langle\{x{:}T'_1 \mid e'_2\}, e'_2[v_2/x], v_2\rangle^l$ by E_Check. Otherwise, $\langle\{x{:}T''_1 \mid e'_1\} \Rightarrow \{x{:}T''_2 \mid e'_2\}\rangle^l\, v_2 \longrightarrow \langle T''_1 \Rightarrow \{x{:}T''_2 \mid e'_2\}\rangle^l\, v_2$ by E_Forget.

(T_TAbs): $\emptyset \vdash \Lambda\alpha.\ e' : \forall\alpha.\ T$. In this case, $\Lambda\alpha.\ e'$ is a result.

(T_TApp): $\emptyset \vdash e_1\ T_2 : T_1[T_2/\alpha]$; by inversion, $\emptyset \vdash e_1 : \forall\alpha.\ T_1$ and $\emptyset \vdash T_2$. By the IH on the first derivation, e_1 steps or is a result. If $e_1 \longrightarrow e'_1$, then $e_1\ T_2 \longrightarrow e'_1\ T_2$ by E_Compat. If $e_1 = \Uparrow l$, then $\Uparrow l\ T_2 \longrightarrow \Uparrow l$ by E_Blame.

If $e_1 = v_1$, then we know that $v_1 = \Lambda\alpha.\ e'_1$ by canonical forms (Lemma 20). We can see $(\Lambda\alpha.\ e'_1)\ T_2 \longrightarrow e'_1[T_2/\alpha]$ by E_TBeta.

(T_Cast): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : T_1 \to T_2$. In this case, $\langle T_1 \Rightarrow T_2\rangle^l$ is a result.

(T_Check): $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l : \{x{:}T \mid e_1\}$; by inversion, $\emptyset \vdash e_2 : \mathsf{Bool}$. By the IH, either $e_2 \longrightarrow e'_2$ steps or $e_2 = r_2$. In the first case, $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l \longrightarrow \langle\{x{:}T \mid e_1\}, e'_2, v\rangle^l$ by E_Compat. In the second case, either $r_2 = \Uparrow l$ or $r_2 = v_2$. If we have blame, then the entire term steps by E_Blame. If we have a value, then we know that v_2 is either true or false, since it's typed at Bool. If it's true, we step by E_OK. Otherwise we step by E_Fail.

(T_Blame): $\emptyset \vdash \Uparrow l : T$. In this case, $\Uparrow l$ is a result.

(Conv): $\emptyset \vdash e : T'$; by inversion, $\emptyset \vdash e : T$. By the IH, we see that $e \longrightarrow e'$ or $e = r$.

(Exact): $\emptyset \vdash v : \{x{:}T \mid e\}$. Here, v is a result by assumption.

(Forget): $\emptyset \vdash v : T$. Again, v is a result by assumption.

Theorem 23 (Preservation). *If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.*

Proof. By induction on the typing derivation.
(T_Var): Contradictory—we can't have $\emptyset \vdash x : T$.
(T_Const): $\emptyset \vdash k : \mathrm{ty}\,(k)$. Contradictory—values don't step.
(Op): $\emptyset \vdash \mathrm{op}\,(e_1, \ldots, e_n) : \sigma(T)$. By cases on the step taken:

> (E_Reduce/E_Op): $\mathrm{op}\,(v_1, \ldots, v_n) \longrightarrow [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n)$. This case is by assumption.
> (E_Blame): $e_i = \Uparrow l$, and everything to its left is a value. By context and type well formedness (Lemma 21), $\emptyset \ \vdash \ \sigma(T)$. So by T_Blame, $\emptyset \vdash \Uparrow l : \sigma(T)$.
> (E_Compat): Some $e_i \longrightarrow e'_i$. By the IH and T_Op, using T_Conv to show that $\sigma(T) \equiv \sigma'(T)$.

(T_Abs): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \rightarrow T_2)$. Contradictory—values don't step.
(T_App): $\emptyset \vdash e_1\ e_2 : T'_2[e_2/x]$, with $\emptyset \vdash e_1 : (x{:}T'_1 \rightarrow T'_2)$ and $\emptyset \vdash e_2 : T'_1$, by inversion. By cases on the step taken.

> (T_Reduce/T_Beta): $(\lambda x{:}T_1.\ e_{12})\, v_2 \longrightarrow e_{12}[v_2/x]$. First, we have $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T'_1 \rightarrow T'_2)$. By inversion for lambdas (Lemma 16), $x{:}T_1 \vdash e_{12} : T_2$. Moreover, $x{:}T_1 \rightarrow T_2 \equiv x{:}T'_1 \rightarrow T'_2$, which means $T_2 \equiv T'_2$.
> By substitution, $\emptyset \vdash e_{12}[v_2/x] : T_2[v_2/x]$. We then see that $T_2[v_2/x] \equiv T'_2[v_2/x]$, so T_Conv completes this case.
> (T_Reduce/T_Refl): $\langle T \Rightarrow T\rangle^l\, v_2 \longrightarrow v_2$. By cast inversion (Lemma 17), $\_{:}T \rightarrow T \equiv x{:}T'_1 \rightarrow T'_2$ and $\emptyset \ \vdash \ T$. In particular, this means that $T \equiv T'_2$ and $T \equiv T'_1$. By substitutivity of conversion, $T[v_2/x] \equiv T'_2[v_2/x]$. Since T is closed, we really know that $T \equiv T'_2[v_2/x]$.
> We already have $\emptyset \vdash v_2 : T'_1$; by T_Conv, $\emptyset \vdash v_2 : T$. By another application of T_Conv, $\emptyset \vdash v : T'_2[v_2/x]$.
> (T_Reduce/T_Forget): $\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v \longrightarrow \langle T_1 \Rightarrow T_2\rangle^l\, v$. We restate the typing judgment and its inversion:
>
> $$\begin{array}{l} \emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v_2 : T'_2[v_2/y] \\ \emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l : (y{:}T'_1 \rightarrow T'_2) \\ \emptyset \vdash v_2 : T'_1 \end{array}$$
>
> By cast inversion (Lemma 17), we know that $\emptyset \ \vdash \ T_1$ and $\emptyset \ \vdash \ T_2$—as well as $\_{:}\{x{:}T_1 \mid e\} \rightarrow T_2 \equiv y{:}T'_1 \rightarrow T'_2$ and $\{x{:}T_1 \mid e\} \parallel T_2$. We can take apart this conversion, finding $\{x{:}T_1 \mid e\} \equiv T'_1$ and $T_2 \equiv T'_2$. Then by T_Conv, $\emptyset \vdash v_2 : \{x{:}T_1 \mid e\}$; by T_Forget, $\emptyset \vdash v_2 : T_1$.
> By T_Cast, we have $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : y{:}T_1 \rightarrow T_2$, with $T_1 \parallel T_2$ iff $\{x{:}T_1 \mid e\} \parallel T_2$. (Note, however, that y does not appear in T_2—we write it to clarify the substitutions below.)
> By T_App, we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l\, v_2 : T_2[v_2/y]$. We have $T_2 \equiv T'_2$, so $T_2[v/y] \equiv T'_2[v_2/y]$. Thus, we conclude this case with T_Conv.
> (T_Reduce/T_PreCheck): $\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v \longrightarrow \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, (\langle T_1 \Rightarrow T_2\rangle^l\, v)$. We restate the typing judgment and its inversion:
>
> $$\begin{array}{l} \emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v_2 : T'_2[v_2/y] \\ \emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l : y{:}T'_1 \rightarrow T'_2 \\ \emptyset \vdash v : T'_1 \end{array}$$

By cast inversion (Lemma 17), $\emptyset \vdash T_1$ and $\emptyset \vdash \{x{:}T_2 \mid e\}$, and $y{:}T_1 \to \{x{:}T_2 \mid e\} \equiv y{:}T_1' \to T_2'$. Also, $T_1 \parallel \{x{:}T_2 \mid e\}$.
By inversion on $\emptyset \vdash \{x{:}T_2 \mid e\}$, we find $\emptyset \vdash T_2$. Next, $T_1 \parallel T_2$ iff $T_1 \parallel \{x{:}T_2 \mid e\}$. Now by T_Cast, we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : y{:}T_1 \to T_2$. Note, however, that y doesn't occur in T_2.
We can take the convertible function types and see that their parts are convertible: $T_1 \equiv T_1'$ and $\{x{:}T_2 \mid e\} \equiv T_2'$. Using the first conversion, we find $\emptyset \vdash v_2 : T_1$ by T_Conv. By T_App, $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l\, v_2 : T_2[v_2/y]$, where $T_2[v_2/y] = T_2$.
By C_Refl and C_RefineR, $T_2 \parallel \{x{:}T_2 \mid e\}$. We have well formedness derivations for both types, as well, so $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l : y{:}T_2 \to \{x{:}T_2 \mid e\}$ by T_Cast. Again, y does not appear in e or T_2. By T_App, we have $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\,(\langle T_1 \Rightarrow T_2\rangle^l\, v_2) : \{x{:}T_2 \mid e\}[\langle T_1 \Rightarrow T_2\rangle^l\, v_2/y]$.
Since y isn't in $\{x{:}T_2 \mid e\}$, we can see:

$$\{x{:}T_2 \mid e\}[\langle T_1 \Rightarrow T_2\rangle^l\, v_2/y] = \{x{:}T_2 \mid e\} = \{x{:}T_2 \mid e\}[v_2/y]$$

Now, by substitutivity of conversion:

$$\{x{:}T_2 \mid e\}[v_2/y] \equiv T_2'[v_2/y]$$

We can now apply T_Conv to find $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\,(\langle T_1 \Rightarrow T_2\rangle^l\, v_2) : T_2'[v_2/y]$.
(T_Reduce/T_Check): $\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v \longrightarrow \langle\{x{:}T \mid e\}, e[v/x], v\rangle^{l'}$. We restate the typing judgment with its inversion:

$$\begin{array}{l}\emptyset \vdash \langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v_2 : T_2'[v/y]\\ \emptyset \vdash \langle T \Rightarrow \{x{:}T \mid e\}\rangle^l : y{:}T_1' \to T_2'\\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma 17), $\emptyset \vdash \{x{:}T \mid e\}$ and $\emptyset \vdash T$. Moreover, $y{:}T \to \{x{:}T \mid e\} \equiv y{:}T_1' \to T_2'$, where y doesn't occur in $\{x{:}T \mid e\}$. This means that $T \equiv T_1'$ and $\{x{:}T \mid e\} \equiv T_2'$.
Using T_Conv with the first conversion shows $\emptyset \vdash v_2 : T$. By inversion on $\emptyset \vdash \{x{:}T \mid e\}$, we see $x{:}T \vdash e : \mathsf{Bool}$. By term substitution (Lemma 11), we find $\emptyset \vdash e[v_2/x] : \mathsf{Bool}$. Finally, $e[v_2/x] \longrightarrow^* e[v_2/x]$ by reflexivity.
T_Check (with WF_Empty) shows $\emptyset \vdash \langle\{x{:}T \mid e\}, e[v_2/x], v_2\rangle^l : \{x{:}T \mid e\}$. By substitutivity of conversion, $\{x{:}T \mid e\}[v_2/y] \equiv T_2[v_2/y]$. Since y doesn't occur in $\{x{:}T \mid e\}$, we know that $\{x{:}T \mid e\}[v/y] = \{x{:}T \mid e\}$. Now $\emptyset \vdash \langle\{x{:}T \mid e\}, e[v_2/x], v_2\rangle^l : T_2[v_2/y]$ by T_Conv.
(T_Reduce/T_Fun): $\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v_2 \longrightarrow \lambda x{:}T_{21}.\,(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)))$. We restate the typing judgment with its inversion:

$$\begin{array}{l}\emptyset \vdash \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v_2 : T_2'[v_2/y]\\ \emptyset \vdash \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l : (y{:}T_1' \to T_2')\\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion on the first derivation:

$$\begin{array}{l}\emptyset \vdash x{:}T_{11} \to T_{12} \\ \emptyset \vdash x{:}T_{21} \to T_{22} \\ x{:}T_{11} \to T_{12} \parallel x{:}T_{21} \to T_{22} \\ \_{:}(x{:}T_{11} \to T_{12}) \to (x{:}T_{21} \to T_{22}) \equiv y{:}T_1' \to T_2'\end{array}$$

By inversion of this last:

$$x{:}T_{11} \to T_{12} \equiv T_1' \qquad x{:}T_{21} \to T_{22} \equiv T_2'$$

So by T_Conv, we have $\emptyset \vdash v_2 : x{:}T_{11} \to T_{12}$. By weakening (Lemma 9), $x{:}T_{21} \vdash v_2 : x{:}T_{11} \to T_{12}$.
By inversion of the well formedness of the function types:

$$\begin{array}{l}\emptyset \vdash T_{11}\ x{:}T_{11} \vdash T_{12} \\ \emptyset \vdash T_{21}\ x{:}T_{21} \vdash T_{22}\end{array}$$

By weakening (Lemma 9), we find $x{:}T_{21} \vdash T_{11}$ and $x{:}T_{21} \vdash T_{21}$. By inversion of compatibility:

$$T_{11} \parallel T_{21} \qquad T_{12} \parallel T_{22}$$

By T_Cast, $x{:}T_{21} \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l : (\_{:}T_{21} \to T_{11})$. By T_App and T_Var, we can see $x{:}T_{21} \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l\, x : T_{11}[x/\_] = T_{11}$. Again by T_App, we have $x{:}T_{21} \vdash v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x) : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x]$. By weakening (Lemma 9) and substitution (Lemma 11), we have the following two derivations:

$$x{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \qquad \emptyset \vdash T_{22}[x/x]$$

By T_Cast (and the fact that compatibility is preserved by substitution):

$$x{:}T_{21} \vdash \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l : (y{:}T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \to T_{22})$$

Noting that y is free here. By T_App:

$$x{:}T_{21} \vdash \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)) : T_{22}[v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)/y] = T_{22}$$

Finally, by T_Abs:

$$\emptyset \vdash \lambda x{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)) : x{:}T_{21} \to T_{22}$$

Since y isn't in $x{:}T_{21} \to T_{22}$, we can see that $x{:}T_{21} \to T_{22} = (x{:}T_{21} \to T_{22})[v_2/y]$. Using this fact with substitutivity of conversion, we find $x{:}T_{21} \to T_{22} \equiv T_2'[v_2/y]$. So—finally—by T_Conv we have:

$$\emptyset \vdash \lambda x{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)) : T_2'[v_2/y]$$

(T_Reduce/T_Forall): $\langle \forall \alpha.\ T_1 \Rightarrow \forall \alpha.\ T_2 \rangle^l\, v_2 \longrightarrow (\Lambda \alpha.\ \langle T_1 \Rightarrow T_2 \rangle^l\, (v\, \alpha))$ We restate the typing and its inversion:

$$\begin{array}{l} \emptyset \vdash \langle \forall \alpha.\ T_1 \Rightarrow \forall \alpha.\ T_2 \rangle^l\, v_2 : T_2'[v_2/x] \\ \emptyset \vdash \langle \forall \alpha.\ T_1 \Rightarrow \forall \alpha.\ T_2 \rangle^l : x{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1' \end{array}$$

By cast inversion (Lemma 17):

$$\emptyset \ \vdash \ \forall \alpha.\ T_1$$

$$\emptyset \ \vdash \ \forall \alpha.\ T_2$$

$$\forall \alpha.\ T_1 \parallel \forall \alpha.\ T_2$$

$$\_{:}(\forall \alpha.\ T_1) \to (\forall \alpha.\ T_2) \equiv x{:}T_1' \to T_2'$$

By inversion of this last $\forall \alpha.\ T_1 \equiv T_1'$ and $\forall \alpha.\ T_2 \equiv T_2'$. By T_Conv, $\emptyset \vdash v_2 : \forall \alpha.\ T_1$. By type variable weakening, WF_TVar, and T_TApp, we have:

$$\alpha \vdash v_2\, \alpha : T_1[\alpha/\alpha] = T_1$$

By inversion of the universal types's well formedness and compatibility:

$$\alpha \ \vdash \ T_1$$

$$\alpha \ \vdash \ T_2$$

$$T_1 \parallel T_2$$

So by T_Cast, $\alpha \vdash \langle T_1 \Rightarrow T_2 \rangle^l : (x{:}T_1 \to T_2)$, noting that x doesn't occur in T_2. By T_App, $\alpha \vdash \langle T_1 \Rightarrow T_2 \rangle^l\, (v_2\, \alpha) : T_2[v_2\, \alpha/x] = T_2$. By T_TAbs, $\emptyset \vdash \Lambda \alpha.\ (\langle T_1 \Rightarrow T_2 \rangle^l\, (v\, \alpha)) : \forall \alpha.\ T_2$.

We know that $\forall \alpha.\ T_2 \equiv T_2'$, so by type variable substitutivity of type equivalence, $(\forall \alpha.\ T_2)[v_2/x] \equiv T_2'[v_2/x]$. Since x isn't in $\forall \alpha.\ T_2$, we know that $\forall \alpha.\ T_2 \equiv T_2'[v_2/x]$. Now we can see by T_Conv that $\emptyset \vdash \Lambda \alpha.\ (\langle T_1 \Rightarrow T_2 \rangle^l\, (v\, \alpha)) : T_2'[v_2/x]$.

(T_Compat): $E\,[e] \longrightarrow E\,[e']$ when $e \longrightarrow e'$ By cases on E:

(E = [] e_2, $e_1 \longrightarrow e_1'$): By the IH and T_App.

($E = v_1\,[]$, $e_2 \longrightarrow e_2'$): By the IH, T_App, and T_Conv, since $T_2[e_2/x] \equiv T_2[e_2'/x]$ by substitutivity.

(T_Blame): $E\,[\Uparrow l] \longrightarrow \Uparrow l$ $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 21), we know that $\emptyset \ \vdash \ T$. We then have $\emptyset \vdash \Uparrow l : T$ by T_Blame.

(T_TAbs): $\emptyset \vdash \Lambda \alpha.\ e : \forall \alpha.\ T$. This case is contradictory—values don't step.

(T_TApp): $\emptyset \vdash e\ T : T'[T/\alpha]$. By cases on the step taken.

(T_Reduce/T_TBeta): $(\Lambda\alpha.\ e')\ T \longrightarrow e'[T/\alpha]$ We restate the typing derivation and its inversion:

$$\emptyset \vdash (\Lambda\alpha.\ e')\ T : T'[T/\alpha]$$
$$\emptyset \vdash \Lambda\alpha.\ e' : \forall\alpha.\ T'$$
$$\emptyset \vdash T$$

By big lambda inversion (Lemma 19):

$$\alpha \vdash v : T''$$

$$\forall\alpha.\ T'' \equiv \forall\alpha.\ T'$$

By inversion of this last, $T'' = T'$.
By type variable substitution (Lemma 14), $\emptyset \vdash e'[T/\alpha] : T''[T/\alpha]$. By substitutivity of type equivalence, $T''[T/\alpha] \equiv T'[T/\alpha]$. T_Conv gives us $\emptyset \vdash e'[T/\alpha] : T'[T/\alpha]$ as desired.
(T_Compat): $E\,[e] \longrightarrow E\,[e']$, where $E = [\,]\ T$. By the IH and T_TApp.
(T_Blame): $E\,[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 21), we know that $\emptyset \vdash T$. So we see $\emptyset \vdash \Uparrow l : T$ by T_Blame.

(T_Cast): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : T_1 \to T_2$. This case is contradictory—values don't step.
(T_Check): $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l : \{x{:}T \mid e_1\}$. By cases on the step taken.

(T_Reduce/T_OK): $\langle\{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \longrightarrow v$. By inversion, $\emptyset \vdash v : T$ and $\emptyset \vdash \{x{:}T \mid e\}$; we also have $e_1[v/x] \longrightarrow^* \mathsf{true}$. By WF_Empty and the assumption that $e[v/x] \longrightarrow^* \mathsf{true}$, we can find $\emptyset \vdash v : \{x{:}T \mid e\}$ by T_Exact.
(T_Reduce/T_Fail): $\langle\{x{:}T \mid e_1\}, \mathsf{false}, v\rangle^l \longrightarrow \Uparrow l$ We have $\emptyset \vdash \{x{:}T \mid e\}$ by inversion. By WF_Empty and T_Blame, $\emptyset \vdash \Uparrow l : \{x{:}T \mid e\}$.
(T_Compat): $E\,[e] \longrightarrow E\,[e']$, where $E = \langle\{x{:}T \mid e_1\}, [\,], v\rangle^l$. By the IH on e, we know that $\emptyset \vdash e' : \mathsf{Bool}$. We still have $\emptyset \vdash \{x{:}T \mid e_1\}$ and $\emptyset \vdash v : T$ from our original derivation. Since $e_1[v/x] \longrightarrow^* e$ and $e \longrightarrow e'$, then $e_1[v/x] \longrightarrow^* e'$. Therefore, $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e', v\rangle^l : \{x{:}T \mid e_1\}$ by T_Check.
(T_Blame): $E\,[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 21), we know that $\emptyset \vdash T$. So $\emptyset \vdash \Uparrow l : T$ by T_Blame.

(T_Blame): $\emptyset \vdash \Uparrow l : T$. This case is contradictory—blame doesn't step.
(T_Conv): $\emptyset \vdash e : T'$; by inversion we have $\emptyset \vdash e : T$ and $T \equiv T'$ and $\emptyset \vdash T'$ (and, trivially, $\vdash \emptyset$). By the IH on the first derivation, we know that $\emptyset \vdash e' : T$. By T_Conv, we can see that $\emptyset \vdash e' : T'$.
(Exact): $\emptyset \vdash v : \{x{:}T \mid e\}$. This case is contradictory—values don't step.
(Forget): $\emptyset \vdash v : T$. This case is contradictory—values don't step.

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

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

$$R_{T,\theta,\delta} = \{(r_1, r_2) \mid r_1 \sim r_2 : T;\theta;\delta\}$$

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

$$\begin{array}{rcl}
B \simeq B : *;\theta;\delta & & \\
\alpha \simeq \alpha : *;\theta;\delta & & \\
x{:}T_{11} \to T_{12} \simeq x{:}T_{21} \to T_{22} : *;\theta;\delta & \iff & T_{11} \simeq T_{21} : *;\theta;\delta \wedge \\
 & & \forall v_1 \sim v_2 : T_{11};\theta;\delta. \\
 & & \quad T_{12} \simeq T_{22} : *;\theta;\delta[v_1, v_2/x] \\
\forall\alpha.\ T_1 \simeq \forall\alpha.\ T_2 : *;\theta;\delta & \iff & \forall R, T_1', T_2'.\ T_1 \simeq T_2 : *;\theta[\alpha \mapsto R, T_1', T_2'];\delta \\
\{x{:}T_1 \mid e_1\} \simeq \{x{:}T_2 \mid e_2\} : *;\theta;\delta & \iff & T_1 \simeq T_2 : *;\theta;\delta \wedge \\
\multicolumn{3}{r}{\forall v_1 \sim v_2 : T_1;\theta;\delta.\ \theta_1(\delta_1(e_1))[v_1/x] \simeq \theta_2(\delta_2(e_2))[v_2/x] : \mathsf{Bool};\theta;\delta}
\end{array}$$

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

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

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. In fact, after cutting out universal types and restricting refinements to base types, it's possible to simplify the operational semantics and to do away with the T_Forget rule. We don't give the proof here because it is subsumed by type soundness in $\mathrm{F_H}$.

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 logically related terms are contextually equivalent, this property means that upcasts can be eliminated without changing the meaning of a program.

We begin by defining a relation on closed results, $r_1 \sim r_2 : T; \theta; \delta$, as a fixpoint on types; we then extend it to a relation on closed expressions, $e_1 \simeq e_2 : T; \theta; \delta$. 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 a pair 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 $\simeq$ 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, T_{11} on the second line could be T_{21} in the function type case), 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

[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.

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 $x{:}T_1 \to T_2$ must take 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 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.)

[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!

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

Lemma 24 (Result LR closed under parallel reduction). *If $r_1 \Rrightarrow r_1'$ and $r_2 \Rrightarrow r_2'$ then $r_1 \sim r_2 : T;\theta;\delta$ iff $r_1' \sim r_2' : T;\theta;\delta$.*

Proof. By induction on T.
$\underline{T = B}$: By cotermination at constants and blame.
$\underline{T = \alpha}$: By assumption.
$\underline{T = x{:}T_1 \to T_2}$: By the IH on T_2 and P_App.
$\underline{T = \forall\alpha.\ T'}$: By the on T' and P_TApp.
$\underline{T = \{x{:}T' \mid e\}}$: By the IH, substitutivity of parallel reduction (Lemma 2), and cotermination.

As a corollary, $\simeq$ is also closed under parallel reduction.

Lemma 25 (Type LR closed under parallel reduction). *If $T_1 \Rrightarrow T_1'$ and $T_2 \Rrightarrow T_2'$ then $T_1 \simeq T_2 : *;\theta;\delta$ iff $T_1' \simeq T_2' : *;\theta;\delta$.*

Proof. By induction on the (common) shape of T_1 and T_2.
$\underline{T_1 = T_2 = B}$: We have $\Rrightarrow$ only by reflexivity, so immediate.
$\underline{T_1 = T_2 = \alpha}$: We have $\Rrightarrow$ only by reflexivity, so immediate.
$\underline{T_1 = x{:}T_{11} \to T_{12} \text{ and } T_2 = x{:}T_{21} \to T_{22}}$: First, inversion gives us piece-wise parallel reductions. By the IH on T_{11} and T_{21}, then by the IH on T_{12} and T_{22}.
$\underline{T_1 = \forall\alpha.\ T_{11} \text{ and } T_2 = \forall\alpha.\ T_{21}}$: First, inversion gives us $T_{11} \Rrightarrow T_{11}'$ and $T_{21} \Rrightarrow T_{21}'$; by the IH for these types.
$\underline{T_1 = \{x{:}T_{11} \mid e_1\} \text{ and } T_2 = \{x{:}T_{21} \mid e_2\}}$: First, inversion gives us parallel reductions for the refined type. There, by the IH. We get the second conditions by term closure under parallel reduction (Lemma 24).

Lemma 26 (Term compositionality). *If $\delta_1(e) \longrightarrow^* e_1$ and $\delta_2(e) \longrightarrow^* e_2$ then $r_1 \sim r_2 : T;\theta;\delta[e_1, e_2/x]$ iff $r_1 \sim r_2 : T[e/x];\theta;\delta$.*

Proof. By induction on the (simple) structure of T, proving both directions simultaneously.

We treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it's the same easy proof in all cases: $\Uparrow l \sim \Uparrow l : T;\theta;\delta$ irrespective of T and δ. So for the rest of proof, we know $r_1 = v_1$ and $r_2 = v_2$.
$\underline{(T = B)}$: Note that $B[e/x] = B$. So we must show that $v_1 \sim v_2 : B;\theta;\delta$ iff $v_1 \sim v_2 : B;\theta;\delta[e_1, e_2/x]$. In either direction, $v_1 = v_2 = k \in \mathcal{K}_B$, irrespective of δ.
$\underline{(T = \alpha)}$: Again, $\alpha[e/x] = \alpha$. So we must show that $v_1 \sim v_2 : \alpha;\theta;\delta$ iff $v_1 \sim v_2 : \alpha;\theta;\delta[e_1, e_2/x]$. In either direction, we must have $\alpha \mapsto R, T_1, T_2 \in \theta$ and $v_1 \mathrel{R} v_2$, which holds irrespective of δ.
$\underline{(T = y{:}T_1 \to T_2)}$: There are two cases:

$\underline{(\Rightarrow)}$: Given $v_1 \sim v_2 : y{:}T_1 \to T_2;\theta;\delta[e_1, e_2/x]$ and reductions for e as above, we must show that $v_1 \sim v_2 : (y{:}T_1 \to T_2)[e/x];\theta;\delta$.

Let $v_1' \sim v_2' : T_1[e/x];\theta;\delta$. We want to prove $v_1\, v_1' \simeq v_2\, v_2' : T_2[e/x];\theta;\delta[v_1', v_2'/y]$. By the IH on T_1, $v_1' \sim v_2' : T_1;\theta;\delta[e_1, e_2/x]$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : T_2;\theta;\delta[e_1, e_2/x][v_1', v_2'/y]$. These normalize to $r_1' \sim r_2' : T_2;\theta;\delta[e_1, e_2/x][v_1', v_2'/y]$. By the IH on T_2, $r_1' \simeq r_2' : T_2[e/x];\theta;\delta[v_1', v_2'/y]$. By expansion, we find $v_1\, v_1' \simeq v_2\, v_2' : T_2[e/x];\theta;\delta[v_1', v_2'/y]$. Therefore $v_1 \sim v_2 : (x{:}T_1 \to T_2)[e/x];\theta;\delta$.

$\underline{(\Leftarrow)}$: This case is similar: given $v_1 \sim v_2 : (y{:}T_1 \to T_2)[e/x];\theta;\delta$ and $\delta_1(e) \longrightarrow^* e_1$ and $\delta_2(e) \longrightarrow^* e_2$, we wish to show that $v_1 \sim v_2 : y{:}T_1 \to T_2;\theta;\delta[e_1, e_2/x]$.

Let $v_1' \sim v_2' : T_1;\theta;\delta[e_1, e_2/x]$. We must show that $v_1\, v_1' \simeq v_2\, v_2' : T_2;\theta;\delta[e_1, e_2/x][v_1', v_2'/y]$. By the IH on T_1, we know that $v_1' \sim v_2' : T_1[e/x];\theta;\delta$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : T_2[e/x];\theta;\delta[v_1', v_2'/y]$. These normalize to $r_1' \sim r_2' : T_2[e/x];\theta;\delta[v_1', v_2'/y]$. By the IH on T_2, we have $r_1' \sim r_2' : T_2;\theta;\delta[v_1', v_2'/y][e_1, e_2/x]$. By expansion, we have $v_1\, v_1' \simeq v_2\, v_2' : T_2;\theta;\delta[v_1', v_2'/y][e_1, e_2/x]$. Since $\delta[v_1', v_2'/y][e_1, e_2/x] = \delta[e_1, e_2/x][v_1', v_2'/y]$, we can see $v_1 \sim v_2 : x{:}T_1 \to T_2;\theta;\delta[v_1', v_2'/y][e_1, e_2/x]$.

$\underline{(T = \forall\alpha.\ T')}$: There are two cases:

$\underline{(\Rightarrow)}$: Given $v_1 \sim v_2 : \forall\alpha.\ T';\theta;\delta[e_1, e_2/x]$ and reductions for e as above, we must show that $v_1 \sim v_2 : (\forall\alpha.\ T')[e/x];\theta;\delta$.

Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\ T_1 \simeq v_2\ T_2 : T';\theta[\alpha \mapsto R, T_1, T_2];\delta[e_1, e_2/x]$. By definition, these normalize to $r_1' \sim r_2' : T';\theta[\alpha \mapsto R, T_1, T_2];\delta[e_1, e_2/x]$. By the IH on T', $r_1' \sim r_2' : T'[e/x];\theta[\alpha \mapsto R, T_1, T_2];\delta$. We can therefore conclude by expansion that $v_1\ T_1 \simeq v_2\ T_2 : T'[e/x];\theta[\alpha \mapsto R, T_1, T_2];\delta$, so $v_1 \sim v_2 : (\forall\alpha.\ T')[e/x];\theta;\delta$.

$\underline{(\Leftarrow)}$: This case is similar: given $v_1 \sim v_2 : (\forall\alpha.\ T')[e/x];\theta;\delta$ and $\delta_1(e) \longrightarrow^* e_1$ and $\delta_2(e) \longrightarrow^* e_2$, we wish to show that $v_1 \sim v_2 : \forall\alpha.\ T';\theta;\delta[e_1, e_2/x]$.

Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\ T_1 \simeq v_2\ T_2 : T'[e/x];\theta[\alpha \mapsto R, T_1, T_2];\delta$. These normalize to $r_1' \sim r_2' : T'[e/x];\theta[\alpha \mapsto R, T_1, T_2];\delta$. By the IH on T' and expansion, we know that $v_1\ T_1 \simeq v_2\ T_2 : T';\theta[\alpha \mapsto R, T_1, T_2];\delta[e_1, e_2/x]$.

$\underline{(T = \{y{:}T' \mid e'\})}$: We show both directions simultaneously.

By the IH for T', $v_1 \sim v_2 : T';\theta;\delta[e_1, e_2/x]$ iff $v_1 \sim v_2 : T'[e/x];\theta;\delta$. Furthermore, by Lemmas 4 and 2, $\delta_1(e'[e/x])[v_1/y] \Rrightarrow^* \delta_1(e')[e_1/x][v_1/y]$ and, similarly, $\delta_2(e'[e/x])[v_2/y] \Rrightarrow^* \delta_2(e')[e_2/x][v_2/y]$. So these terms coterminate at true, that is, $\delta_1(e'[e/x])[v_1/y] \longrightarrow^* \mathsf{true}$ iff $\delta_1(e')[e_1/x][v_1/y] \longrightarrow^* \mathsf{true}$ and similarly for $\delta_2(e')[e_2/x][v_2/y]$ and $\delta_2(e'[e/x])[v_2/y]$.

Lemma 27 (Term Weakening/Strengthening). *If $x \notin T$, then $r_1 \sim r_2 : T;\theta;\delta[e_1, e_2/x]$ if and only if $r_1 \sim r_2 : T;\theta;\delta$.*

Proof. Similar to Lemma 26.

Lemma 28 (Type compositionality). $r_1 \sim r_2 : T;\theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))];\delta$ *iff* $r_1 \sim r_2 : T[T'/\alpha];\theta;\delta$.

Proof. By induction on the (simple) structure of T, proving both directions simultaneously.

We treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it's the same easy proof in all cases: $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ irrespective of T and δ. So for the rest of proof, we know $r_1 = v_1$ and $r_2 = v_2$.

$\underline{(T = B)}$: Note that $B[T'/\alpha] = B$. So we must show that $v_1 \sim v_2 : B; \theta'; \delta$ iff $v_1 \sim v_2 : B; \theta; \delta$. In either direction, $v_1 = v_2 = k \in \mathcal{K}_B$, irrespective of θ.

$\underline{(T = \alpha)}$: We have

$$\begin{aligned} & v_1 \sim v_2 : \alpha; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta \\ \iff & v_1 \; R_{T',\theta,\delta} \; v_2 \\ \iff & v_1 \sim v_2 : T'; \theta; \delta \\ \iff & v_1 \sim v_2 : \alpha[T'/\alpha]; \theta; \delta \end{aligned}$$

$\underline{(T = \alpha' \neq \alpha)}$: Note that $\alpha'[T'/\alpha] = \alpha'$. We have

$$\begin{aligned} & v_1 \sim v_2 : \alpha'; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta \\ \iff & \alpha' \mapsto R, T_1, T_2 \in \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))] \wedge v_1 \; R \; v_2 \\ \iff & \alpha' \mapsto R, T_1, T_2 \in \theta \wedge v_1 \; R \; v_2 \\ \iff & v_1 \sim v_2 : \alpha'; \theta; \delta \end{aligned}$$

$\underline{(T = x{:}T_1 \to T_2)}$: There are two cases:

$\underline{(\Rightarrow)}$:
Given $v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : (x{:}T_1 \to T_2)[T'/\alpha]; \theta; \delta$. Let $v_1' \sim v_2' : T_1[T'/\alpha]; \theta; \delta$. We must show that $v_1\, v_1' \simeq v_2\, v_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. By the IH on T_1, $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x]$. These normalize to $r_1' \sim r_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x]$. Since $x \notin T'$, Lemma 27 gives $R_{T',\theta,\delta} = R_{T',\theta,\delta[v_1',v_2'/x]}$ and so $r_1' \sim r_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta[v_1',v_2'/x]}, \theta_1(\delta_1(T'[v_1'/x])), \theta_2(\delta_2(T'[v_2'/x]))]; \delta[v_1', v_2'/x]$. By the IH on T_2, $r_1' \sim r_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. By expansion, $v_1\, v_1'' \simeq v_2\, v_2'' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$.

$\underline{(\Leftarrow)}$: This case is similar: Given $v_1 \sim v_2 : (x{:}T_1 \to T_2)[T'/\alpha]; \theta; \delta$, we wish to show that $v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Let $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. We must show that $v_1\, v_1' \simeq v_2\, v_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x]$. By the IH on T_1, $v_1' \sim v_2' : T_1[T'/\alpha]; \theta; \delta$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. These normalize to $r_1' \simeq r_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. By the IH on T_2,

$$r_1' \simeq r_2' : T_2[T'/\alpha]; \theta[\alpha \mapsto R_{T',\theta,\delta[v_1',v_2'/x]}, \theta_1(\delta_1(T'[v_1'/x])), \theta_2(\delta_2(T'[v_2'/x]))]; \delta[v_1', v_2'/x].$$

Since $x \notin T'$, Lemma 27 gives

$$r_1' \simeq r_2' : T_2[T'/\alpha]; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x].$$

By expansion, $v_1\, v_1' \simeq v_2\, v_2' : T_2[T'/\alpha]; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x]$.

$\underline{(T = \forall\alpha'.\ T_0)}$: There are two cases:

$\underline{(\Rightarrow)}$: Given $v_1 \sim v_2 : \forall\alpha'.\ T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : \forall\alpha'.\ (T_0[T'/\alpha]); \theta; \delta$. Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\, T_1 \simeq v_2\, T_2 : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))][\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))][\alpha' \mapsto R, T_1, T_2]; \delta$. By the IH, $r_1' \sim r_2' : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By expansion, $v_1\, T_1 \simeq v_2\, T_2 : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. Then, $v_1 \sim v_2 : \forall\alpha'.\ (T_0[T'/\alpha]); \theta; \delta$.

$\underline{(\Leftarrow)}$: This case is similar: given $v_1 \sim v_2 : \forall\alpha'.\ (T_0[T'/\alpha]); \theta; \delta$, we wish to show that $v_1 \sim v_2 : \forall\alpha'.\ T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\, T_1 \simeq v_2\, T_2 : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By the IH, $r_1' \sim r_2' : T_0; \theta[\alpha' \mapsto R, T_1, T_2][\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. By expansion, $v_1\, T_1 \simeq v_2\, T_2 : T_0; \theta[\alpha' \mapsto R, T_1, T_2][\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Then, $v_1 \sim v_2 : \forall\alpha'.\ T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$.

$\underline{(T = \{x{:}T_0 \mid e\}}$: We have

$$
\begin{array}{rl}
 & v_1 \sim v_2 : \{x{:}T_0 \mid e\}; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta \\
\iff & \begin{cases} v_1 \sim v_2 : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta \\ \theta_1(\delta_1(e))[v_1/x][\theta_1(\delta_1(T'))/\alpha] \longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e))[v_2/x][\theta_2(\delta_2(T'))/\alpha] \longrightarrow^* \mathsf{true} \end{cases} \\
\text{(by the IH)} \iff & \begin{cases} v_1 \sim v_2 : T_0[T'/\alpha]; \theta; \delta \\ \theta_1(\delta_1(e))[v_1/x][\theta_1(\delta_1(T'))/\alpha] \longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e))[v_2/x][\theta_2(\delta_2(T'))/\alpha] \longrightarrow^* \mathsf{true} \end{cases} \\
(x \notin T') \iff & \begin{cases} v_1 \sim v_2 : T_0[T'/\alpha]; \theta; \delta \\ \theta_1(\delta_1(e[T'/\alpha]))[v_1/x] \longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e[T'/\alpha]))[v_2/x] \longrightarrow^* \mathsf{true} \end{cases} \\
\iff & v_1 \sim v_2 : \{x{:}T_0[T'/\alpha] \mid e[T'/\alpha]\}; \theta; \delta
\end{array}
$$

Lemma 29 (Convertibility). *If $T_1 \equiv T_2$ and $r_1 \sim r_2 : T_1; \theta; \delta$ then $r_1 \sim r_2 : T_2; \theta; \delta$.*

Proof.

Definition 30 (Complexity of Casts).

$$\begin{cases}
\mathit{cc}(\langle T \Rightarrow T\rangle^{l}) & = 1 \\
\mathit{cc}(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l}) & = \mathit{cc}(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, x/x] \Rightarrow T_{22}\rangle^{l}) \\
& \quad +\mathit{cc}(\langle T_{21} \Rightarrow T_{11}\rangle^{l}) + 1 \\
\mathit{cc}(\langle \forall\alpha.\ T_1 \Rightarrow \forall\alpha.\ T_2\rangle^{l}) & = \mathit{cc}(\langle T_1 \Rightarrow T_2\rangle^{l}) + 1 \\
\mathit{cc}(\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^{l}) & = \mathit{cc}(\langle T_1 \Rightarrow T_2\rangle^{l}) + 1 \\
& \quad \textit{(if } T_2 \neq \{x{:}T_1 \mid e\} \textit{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e'\}\textit{)} \\
\mathit{cc}(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^{l}) & = 1 \\
\mathit{cc}(\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^{l}) & = \mathit{cc}(\langle T_1 \Rightarrow T_2\rangle^{l}) + 2 \\
& \quad \textit{(if } T_1 \neq T_2 \textit{ and } T_1 \textit{ is not a refinement type)}
\end{cases}$$

Lemma 31. *If $T_1 \parallel T_2$, then $T_1[e/x] \parallel T_2$.*

Proof.

Lemma 32. *If $T_1 \parallel T_2$, then $\mathit{cc}(\langle T_1 \Rightarrow T_2\rangle^{l}) > 0$.*

Proof.

Lemma 33 (Cast Reflexivity). *If $\vdash \Gamma$, $T_1 \parallel T_2$, $\Gamma \vdash T_1 \simeq T_1 : *$, and $\Gamma \vdash T_2 \simeq T_2 : *$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^{l} \simeq \langle T_1 \Rightarrow T_2\rangle^{l} : T_1 \to T_2$.*

Proof. By induction on $\mathit{cc}(\langle T_1 \Rightarrow T_2\rangle^{l})$.
$\underline{(T_1 = T_2)}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that $\langle \theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1))\rangle^{l} \sim \langle \theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1))\rangle^{l} : T_1 \to T_1; \theta; \delta$. Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that $\langle \theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1))\rangle^{l}\, v_1 \simeq \langle \theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1))\rangle^{l}\, v_2 : T_1; \theta; \delta[v_1, v_2/z]$ for fresh z. By E_Refl, these normalize to $v_1 \sim v_2 : T_1; \theta; \delta[v_1, v_2/z]$. Lemma 27 finishes the case.
$\underline{(T_1 = x{:}T_{11} \to T_{12} \text{ and } T_2 = x{:}T_{21} \to T_{22} \text{ and } T_1 \neq T_2)}$: Then, we have $T_{11} \parallel T_{21}, T_{21} \parallel T_{22}, \Gamma \vdash T_{11}, \Gamma, x{:}T_{11} \vdash T_{12}$, and $\Gamma, x{:}T_{21} \vdash T_{22}$.

Given $\Gamma \vdash \theta; \delta$, we wish to show that $\theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l})) \sim \theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l})) : (x{:}T_{11} \to T_{12}) \to (x{:}T_{21} \to T_{22}); \theta; \delta$. Let $v_1 \sim v_2 : x{:}T_{11} \to T_{12}; \theta; \delta$. We must show that $\theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l}))\, v_1 \simeq \theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l}))\, v_2 : x{:}T_{21} \to T_{22}; \theta; \delta[v_1, v_2/z]$ for fresh z. Let $v_1' = \theta_1(\delta_1(\lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, x/x] \Rightarrow T_{22}\rangle^{l}\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, x)))))$ and $v_2' = \theta_2(\delta_2(\lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, x/x] \Rightarrow T_{22}\rangle^{l}\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, x)))))$. Since $\theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l}))\, v_1 \longrightarrow v_1'$ and $\theta_2(\delta_2(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l}))\, v_2 \longrightarrow v_2'$, it suffices to show that $v_1' \sim v_2' : x{:}T_{21} \to T_{22}; \theta; \delta[v_1, v_2/z]$. Let $v_1'' \sim v_2'' : T_{21}; \theta; \delta[v_1, v_2/z]$. We must show that $v_1'\, v_1'' \simeq v_2'\, v_2'' : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x]$. Since $v_1'\, v_1'' \longrightarrow \theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_1''/x] \Rightarrow T_{22}\rangle^{l}\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_1''))))$ and $v_2'\, v_2'' \longrightarrow \theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_2''/x] \Rightarrow T_{22}\rangle^{l}\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_2''))))$, it suffices to show that $\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_1''/x] \Rightarrow T_{22}\rangle^{l}\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_1'')))) \simeq \theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_2''/x] \Rightarrow T_{22}\rangle^{l}\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^{l}\, v_2'')))) : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x]$.

Since $\Gamma \vdash T_1 \simeq T_1 : *$ and $\Gamma \vdash T_2 \simeq T_2 : *$, we have $\Gamma \vdash T_{11} \simeq T_{11} : *$ and $\Gamma \vdash T_{21} \simeq T_{21} : *$. Then, by the IH, we have $\Gamma \vdash \langle T_{21} \Rightarrow T_{11}\rangle^{l} \simeq$

$\langle T_{21} \Rightarrow T_{11}\rangle^l : T_{21} \to T_{11}$ and $\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l)) \sim \theta_2(\delta_2(\langle T_{21} \Rightarrow T_{11}\rangle^l)) : T_{21} \to T_{11}; \theta; \delta$. By Lemma 27 and assumption, $\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l))\, v''_1 \simeq \theta_2(\delta_2(\langle T_{21} \Rightarrow T_{11}\rangle^l))\, v''_2 : T_{11}; \theta; \delta[v_1, v_2/z][v''_1, v''_2/z']$ for fresh z'. These normalize to $r_1 \simeq r_2 : T_{11}; \theta; \delta[v_1, v_2/z][v''_1, v''_2/z']$. If $r_1 = r_2 = \Uparrow l$ for some l, then we also have $v_1\, v''_1 \longrightarrow^* \Uparrow l$ and $v_2\, v''_2 \longrightarrow^* \Uparrow l$ and, by expansion, $v_1\, v''_1 \simeq v_2\, v''_2 : T_{22}; \theta; \delta[v_1, v_2/z][v''_1, v''_2/x]$. Otherwise, let $v'''_1 = r_1$ and $v'''_2 = r_2$. By Lemma 27 and definition, $v_1\, v'''_1 \simeq v_2\, v'''_2 : T_{12}; \theta; \delta[v'''_1, v'''_2/x]$. These normalize to $r'_1 \sim r'_2 : T_{12}; \theta; \delta[v'''_1, v'''_2/x]$. If $r'_1 = r'_2 = \Uparrow l'$ for some l', then, again, we have $v_1\, v''_1 \longrightarrow^* \Uparrow l$ and $v_2\, v''_2 \longrightarrow^* \Uparrow l$ and, by expansion, $v_1\, v''_1 \simeq v_2\, v''_2 : T_{22}; \theta; \delta[v_1, v_2/z][v''_1, v''_2/x]$. Otherwise, let $v''''_1 = r'_1$ and $v''''_2 = r'_2$. By Lemma 27 and 26 and the fact that $\langle T_{21} \Rightarrow T_{11}\rangle^l\, v''_1 \longrightarrow^* v'''_1$ and $\langle T_{21} \Rightarrow T_{11}\rangle^l\, v''_2 \longrightarrow^* v'''_2$, we have $v''''_1 \sim v''''_2 : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x]; \theta; \delta[v''_1, v''_2/x]$.

From $\Gamma \vdash T_1 \simeq T_1 : *$, we have $\Gamma, x{:}T_{11} \vdash T_{12} \simeq T_{12} : *$ by definition. Furthermore, it is easy to show that $\Gamma, x{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \simeq T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] : *$.

Since $T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \parallel T_{22}$ by 31, by the IH, $\Gamma, x{:}T_{21} \vdash \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l \simeq \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \to T_{22}$. Since $\Gamma, x{:}T_{21} \vdash \theta; \delta[v''_1, v''_2/x]$ and $x \notin T_{21}, T_{11}$, we have $\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v''_1/x] \Rightarrow T_{22}\rangle^l)) \sim \theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v''_2/x] \Rightarrow T_{22}\rangle^l)) : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \to T_{22}; \theta; \delta[v''_1, v''_2/x]$. By definition, $(\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v''_1/x] \Rightarrow T_{22}\rangle^l)))\, v''''_1 \simeq (\theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v''_2/x] \Rightarrow T_{22}\rangle^l)))\, v''''_2 : T_{22}; \theta; \delta[v''_1, v''_2/x][v''''_1, v''''_2/z'']$ for fresh z''. These normalize to $r''_1 \sim r''_2 : T_{22}; \theta; \delta[v''_1, v''_2/x][v''''_1, v''''_2/z'']$. By expansion and Lemma 27, $v'_1\, v''_1 \simeq v'_2\, v''_2 : T_{22}; \theta; \delta[v_1, v_2/z][v''_1, v''_2/x]$.

$\underline{(T_1 = \forall\alpha.\ T'_1 \text{ and } T_2 = \forall\alpha.\ T'_2 \text{ and } T_1 \neq T_2)}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that $\langle \forall\alpha.\ \theta_1(\delta_1(T'_1)) \Rightarrow \forall\alpha.\ \theta_1(\delta_1(T'_2))\rangle^l \sim \langle \forall\alpha.\ \theta_2(\delta_2(T'_1)) \Rightarrow \forall\alpha.\ \theta_2(\delta_2(T'_2))\rangle^l : (\forall\alpha.\ T'_1) \to (\forall\alpha.\ T'_2); \theta; \delta$. Let $v_1 \sim v_2 : \forall\alpha.\ T'_1; \theta; \delta$. We must show that $\langle \forall\alpha.\ \theta_1(\delta_1(T'_1)) \Rightarrow \forall\alpha.\ \theta_1(\delta_1(T'_2))\rangle^l\, v_1 \simeq \langle \forall\alpha.\ \theta_2(\delta_2(T'_1)) \Rightarrow \forall\alpha.\ \theta_2(\delta_2(T'_2))\rangle^l\, v_2 : (\forall\alpha.\ T'_2); \theta; \delta[v_1, v_2/z]$ for fresh z. Since $\langle \forall\alpha.\ \theta_1(\delta_1(T'_1)) \Rightarrow \forall\alpha.\ \theta_1(\delta_1(T'_2))\rangle^l\, v_1 \longrightarrow \Lambda\alpha.\ \langle \theta_1(\delta_1(T'_1)) \Rightarrow \theta_1(\delta_1(T'_2))\rangle^l\, (v_1\, \alpha)$ and $\langle \forall\alpha.\ \theta_2(\delta_2(T'_1)) \Rightarrow \forall\alpha.\ \theta_2(\delta_2(T'_2))\rangle^l\, v_2 \longrightarrow \Lambda\alpha.\ \langle \theta_2(\delta_2(T'_1)) \Rightarrow \theta_2(\delta_2(T'_2))\rangle^l\, (v_2\, \alpha)$, it suffices to show that $\Lambda\alpha.\ \langle \theta_1(\delta_1(T'_1)) \Rightarrow \theta_1(\delta_1(T'_2))\rangle^l\, (v_1\, \alpha) \sim \Lambda\alpha.\ \langle \theta_2(\delta_2(T'_1)) \Rightarrow \theta_2(\delta_2(T'_2))\rangle^l\, (v_2\, \alpha) : (\forall\alpha.\ T'_2); \theta; \delta[v_1, v_2/z]$. Let R, T''_1, T''_2 be given. We will show that $\Lambda\alpha.\ \langle \theta_1(\delta_1(T'_1)) \Rightarrow \theta_1(\delta_1(T'_2))\rangle^l\, (v_1\, \alpha)\, T''_1 \sim \Lambda\alpha.\ \langle \theta_2(\delta_2(T'_1)) \Rightarrow \theta_2(\delta_2(T'_2))\rangle^l\, (v_2\, \alpha)\, T''_2 : T'_2; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta[v_1, v_2/z]$. Since these normalize to $\theta_1(\delta_1(\langle T'_1 \Rightarrow T'_2\rangle^l[T''_1/\alpha]))\, (v_1\ T''_1)$ and $\theta_2(\delta_2(\langle T'_1 \Rightarrow T'_2\rangle^l[T''_2/\alpha]))\, (v_2\ T''_2)$, we will show $\theta_1(\delta_1(\langle T'_1 \Rightarrow T'_2\rangle^l[T''_1/\alpha]))\, (v_1\ T''_1) \simeq \theta_2(\delta_2(\langle T'_1 \Rightarrow T'_2\rangle^l[T''_2/\alpha]))\, (v_2\ T''_2) : T'_2; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta[v_1, v_2/z]$. By assumption, $v_1\ T''_1 \simeq v_2\ T''_2 : T'_1; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta$. These normalize to $r_1 \sim r_2 : T'_1; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta$. If $r_1 = r_2 = \Uparrow l$ for some l, we have $\Lambda\alpha.\ \langle \theta_1(\delta_1(T'_1)) \Rightarrow \theta_1(\delta_1(T'_2))\rangle^l\, (v_1\, \alpha)\, T''_1 \longrightarrow^* \Uparrow l$ and $\Lambda\alpha.\ \langle \theta_2(\delta_2(T'_1)) \Rightarrow \theta_2(\delta_2(T'_2))\rangle^l\, (v_2\, \alpha)\, T''_2 \longrightarrow^* \Uparrow l$, finishing the case. Otherwise, we have $r_1 = v'_1$ and $r_2 = v'_2$ and $v'_1 \sim v'_2 : T'_1; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta$.

It is easy to show that $\Gamma, \alpha \vdash T'_1 \simeq T'_1 : *$ and $\Gamma, \alpha \vdash T'_2 \simeq T'_2 : *$ from the assumptions $\Gamma, \alpha \vdash T_1 \simeq T_1 : *$ and $\Gamma, \alpha \vdash T_2 \simeq T_2 : *$. Then, by the IH, we have $\Gamma, \alpha \vdash \langle T'_1 \Rightarrow T'_2\rangle^l \simeq \langle T'_1 \Rightarrow T'_2\rangle^l : T'_1 \to T'_2$. Since $\Gamma, \alpha \vdash \theta[\alpha \mapsto R, T''_1, T''_2]; \delta$, we have $\theta_1(\delta_1(\langle T'_1[T''_1/\alpha] \Rightarrow T'_2[T''_1/\alpha]\rangle^l)) \sim \theta_2(\delta_2(\langle T'_1[T''_2/\alpha] \Rightarrow T'_2[T''_2/\alpha]\rangle^l)) : T'_1 \to T'_2; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta$. By definition, $\theta_1(\delta_1(\langle T'_1 \Rightarrow T'_2\rangle^l[T''_1/\alpha]))\, v'_1 \simeq$

$\theta_2(\delta_2(\langle T_1' \Rightarrow T_2'\rangle^l[T_2''/\alpha]))\, v_2' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1', v_2'/z']$ for fresh z'. By Lemma 27, $\theta_1(\delta_1(\langle T_1' \Rightarrow T_2'\rangle^l[T_1''/\alpha]))\, v_1' \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2'\rangle^l[T_2''/\alpha]))\, v_2' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$. The fact that $\theta_1(\delta_1(\langle T_1' \Rightarrow T_2'\rangle^l[T_1''/\alpha]))\,(v_1\ T_1'') \longrightarrow^* \theta_1(\delta_1(\langle T_1' \Rightarrow T_2'\rangle^l[T_1''/\alpha]))\, v_1'$ and $\theta_2(\delta_2(\langle T_1' \Rightarrow T_2'\rangle^l[T_2''/\alpha]))\,(v_2\ T_2'') \longrightarrow^* \theta_2(\delta_2(\langle T_1' \Rightarrow T_2'\rangle^l[T_2''/\alpha]))\, v_2'$ finishes the case.

($T_1 = \{x{:}T_1' \mid e\}$ and $T_1 \neq T_2$ and $T_2 \neq \{y{:}T_1 \mid e'\}$): Given $\Gamma \vdash \theta; \delta$, we wish to show that $\theta_1(\delta_1(\langle\{x{:}T_1' \mid e\} \Rightarrow T_2\rangle^l)) \sim \theta_2(\delta_2(\langle\{x{:}T_1' \mid e\} \Rightarrow T_2\rangle^l)) : \{x{:}T_1' \mid e\} \to T_2; \theta; \delta$. Let $v_1 \sim v_2 : \{x{:}T_1' \mid e\}; \theta; \delta$, that is, $v_1 \sim v_2 : T_1'; \theta; \delta$ and $\theta_1(\delta_1(e))[v_1/x] \longrightarrow^* \mathsf{true}\ \theta_2(\delta_2(e))[v_2/x] \longrightarrow^* \mathsf{true}$. We have to show that $\theta_1(\delta_1(\langle\{x{:}T_1' \mid e\} \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\langle\{x{:}T_1' \mid e\} \Rightarrow T_2\rangle^l))\, v_2 : T_2; \theta; \delta[v_1, v_2/z]$ for fresh z. Since $\theta_1(\delta_1(\langle\{x{:}T_1' \mid e\} \Rightarrow T_2\rangle^l))\, v_1 \longrightarrow \theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l))\, v_1$ and $\theta_2(\delta_2(\langle\{x{:}T_1' \mid e\} \Rightarrow T_2\rangle^l))\, v_2 \longrightarrow \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l))\, v_2$ by E_Forget, it suffices to show that $\theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l))\, v_2 : T_2; \theta; \delta[v_1, v_2/z]$.

Since $\Gamma \vdash T_1 \simeq T_1 : *$, we have $\Gamma \vdash T_1' \simeq T_1' : *$ by definition. We also have $T_1' \parallel T_2$, by inversion. Then, by the IH, $\Gamma \vdash \langle T_1' \Rightarrow T_2\rangle^l \simeq \langle T_1' \Rightarrow T_2\rangle^l : T_1' \to T_2$, and so $\theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l)) \sim \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l)) : T_1' \to T_2; \theta; \delta$. By assumption, $\theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l))\, v_2 : T_2; \theta; \delta[v_1, v_2/z]$, as required.

($T_2 = \{x{:}T_1 \mid e\}$): Given $\Gamma \vdash \theta; \delta$, we wish to show that $\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l)) \sim \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l)) : T_1 \to \{x{:}T_1 \mid e\}; \theta; \delta$. Let $v_1 \sim v_2 : T_1; \theta; \delta$. We have to show that $\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\, v_2 : \{x{:}T_1 \mid e\}; \theta; \delta[v_1, v_2/z]$ for fresh z. Since $\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\, v_1 \longrightarrow \langle\{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l$ and $\theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\, v_2 \longrightarrow \langle\{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l$ by E_Check, it suffices to show that $\langle\{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l \simeq \langle\{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l : \{x{:}T_1 \mid e\}; \theta; \delta[v_1, v_2/z]$.

By the assumption $\Gamma \vdash T_2 \simeq T_2 : *$, we have $T_2 \simeq T_2 : *; \theta; \delta$. By definition, $T_1 \simeq T_1 : *; \theta; \delta$ and $\theta_1(\delta_1(e))[v_1'/x] \simeq \theta_2(\delta_2(e))[v_2'/x] : \mathsf{Bool}; \theta; \delta$ for any $v_1' \sim v_2' : T_1; \theta; \delta$. So, in particular, $\theta_1(\delta_1(e))[v_1/x] \simeq \theta_2(\delta_2(e))[v_2/x] : \mathsf{Bool}; \theta; \delta$. These reduce to $r_1 \sim r_2 : \mathsf{Bool}; \theta; \delta$.

We have three cases:

($r_1 = r_2 = \Uparrow l'$ for some l'): Then, $\langle\{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l \longrightarrow^* \Uparrow l'$ and $\langle\{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l \longrightarrow^* \Uparrow l'$, finishing the case.
($r_1 = r_2 = \mathsf{true}$): Then, $\langle\{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l \longrightarrow^* v_1$ and $\langle\{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l \longrightarrow^* v_2$. By assumption and Lemma 27, $v_1 \sim v_2 : T_1; \theta; \delta[v_1, v_2/z]$, finishing the case.
($r_1 = r_2 = \mathsf{false}$): Then, $\langle\{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l \longrightarrow^* \Uparrow l$ and $\langle\{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l \longrightarrow^* \Uparrow l$, finishing the case.

(T_1 is not a refinement type and $T_2 = \{x{:}T_2' \mid e\}$ and $T_1 \neq T_2'$): Given $\Gamma \vdash \theta; \delta$, we wish to show that $\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l)) \sim \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l)) : T_1 \to \{x{:}T_2' \mid e\}; \theta; \delta$. Let $v_1 \sim v_2 : T_1; \theta; \delta$. We have to show that $\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_2 : \{x{:}T_2' \mid e\}; \theta; \delta[v_1, v_2/z]$ for fresh z. Since $\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_1 \longrightarrow \theta_1(\delta_1(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\,(\theta_1(\delta_1(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_1)$ and $\theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_2 \longrightarrow$

$\theta_2(\delta_2(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\,(\theta_2(\delta_2(\langle T_1 \Rightarrow T_2'\rangle^l))\,v_2)$ by E_PreCheck, it suffices to show that $\theta_1(\delta_1(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\,(\theta_1(\delta_1(\langle T_1 \Rightarrow T_2'\rangle^l))\,v_1) \simeq \theta_2(\delta_2(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\,(\theta_2(\delta_2(\langle T_1 \Rightarrow T_2'\rangle^l))\,v_2) : \{x{:}T_2' \mid e\}; \theta; \delta[v_1, v_2/z]$.

Since $\Gamma \vdash T_2 \simeq T_2 : *$, we have $\Gamma \vdash T_2' \simeq T_2' : *$ by definition. By the IH, $\Gamma \vdash \langle T_1 \Rightarrow T_2'\rangle^l \simeq \langle T_1 \Rightarrow T_2'\rangle^l : T_1 \to T_2'$ and $\Gamma \vdash \langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l \simeq \langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l : T_2' \to \{x{:}T_2' \mid e\}$. (Note that $cc(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l) = cc(\langle T_1 \Rightarrow T_2'\rangle^l) + 2 > cc(\langle T_1 \Rightarrow T_2'\rangle^l)$ and $cc(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l = cc(\langle T_1 \Rightarrow T_2'\rangle^l) + 2 > 1 = cc(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l)$.) Then, $\theta_1(\delta_1(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\,(\theta_1(\delta_1(\langle T_1 \Rightarrow T_2'\rangle^l))\,v_1) \simeq \theta_2(\delta_2(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\,(\theta_2(\delta_2(\langle T_1 \Rightarrow T_2'\rangle^l))\,v_2) : \{x{:}T_2' \mid e\}; \theta; \delta[v_1, v_2/z]$ is easy.

Theorem 34 (Parametricity).

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

Proof. By simultaneous induction on the derivations with case analysis on the last rule used.
(T_Var): Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T; \theta; \delta$, which follows from the assumption.
(T_Const): By the assumption on constants.
(T_Op):
(T_Abs): We have $e = \lambda x{:}T_1.\ e_{12}$, $T = x{:}T_1 \to T_2$, and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(\lambda x{:}T_1.\ e_{12})) \sim \theta_2(\delta_2(\lambda x{:}T_1.\ e_{12})) : (x{:}T_1 \to T_2); \theta; \delta$. Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that $(\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\,v_1 \simeq (\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\,v_2 : T_2; \theta; \delta[v_1, v_2/x]$. Since $(\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\,v_1 \longrightarrow \theta_1(\delta_1(e_{12}))[v_1/x]$ and $(\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\,v_2 \longrightarrow \theta_2(\delta_2(e_{12}))[v_2/x]$, it suffices to show $\theta_1(\delta_1(e_{12}))[v_1/x] \simeq \theta_2(\delta_2(e_{12}))[v_2/x] : T_2; \theta; \delta[v_1, v_2/x]$. By the IH, $\Gamma, x{:}T_1 \vdash e_{12} \simeq e_{12} : T_2$. The fact that $\Gamma, x{:}T_1 \vdash \theta; \delta[v_1, v_2/x]$ finishes the case.
(T_App): We have $e = e_1\,e_2$, $\Gamma \vdash e_1 : x{:}T_1 \to T_2$, $\Gamma \vdash e_2 : T_1$, and $T = T_2[e_2/x]$. Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(e_1\,e_2)) \simeq \theta_2(\delta_2(e_1\,e_2)) : T_2[e_2/x]; \theta; \delta$. By the IH, $\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : x{:}T_1 \to T_2; \theta; \delta$ and $\theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : T_1; \theta; \delta$. These normalize to $r_{11} \sim r_{12} : x{:}T_1 \to T_2; \theta; \delta$ and $r_{21} \simeq r_{22} : T_1; \theta; \delta$, respectively. If $r_{11} = r_{12} = \Uparrow l$ or $r_{21} = r_{22} = \Uparrow l$ for some l, then $\theta_1(\delta_1(e_1\,e_2)) \longrightarrow^* \Uparrow l$ and $\theta_2(\delta_2(e_1\,e_2)) \longrightarrow^* \Uparrow l$, and we are done. Let $r_{i\,j} = v_{i\,j}$. By definition, $v_{11}\,v_{21} \simeq v_{12}\,v_{22} : T_2; \theta; \delta[v_{21}, v_{22}/x]$. These normalize to $r_1' \sim r_2' : T_2; \theta; \delta[v_{21}, v_{22}/x]$. By Lemma 26, $r_1' \sim r_2' : T_2[e_2/x]; \theta; \delta$. By expansion, $\theta_1(\delta_1(e_1\,e_2)) \simeq \theta_2(\delta_2(e_1\,e_2)) : T_2[e_2/x]; \theta; \delta$, as required.
(T_TAbs): We have $e = \Lambda\alpha.\ e_0$, $T = \forall\alpha.\ T_0$, and $\Gamma, \alpha \vdash e_0 : T_0$. Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(\Lambda\alpha.\ e_0)) \sim \theta_2(\delta_2(\Lambda\alpha.\ e_0)) : \forall\alpha.\ T_0; \theta; \delta$. Let R, T_1, T_2 be given. We must show that $\theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 \simeq \theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta$. Since $\theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 \longrightarrow \theta_1(\delta_1(e_0))[T_1/\alpha]$ and $\theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 \longrightarrow \theta_2(\delta_2(e_0))[T_2/\alpha]$, it suffices to show that $\theta_1(\delta_1(e_0))[T_1/\alpha] \simeq \theta_2(\delta_2(e_0))[T_2/\alpha] : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta$. Since $\Gamma, \alpha \vdash \theta[\alpha \mapsto R, T_1, T_2]; \delta$, the IH, which gives $\Gamma, \alpha \vdash e_0 \simeq e_0 : T_0$, finishes the case.
(T_TApp): We have $e = e_1\ T_2$, $\Gamma \vdash e_1 : \forall\alpha.\ T_1$, $\Gamma \vdash\ T_2$, and $T = T_0[T_2/\alpha]$. Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : T_0[T_2/\alpha]; \theta; \delta$.

By the IH, $\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : \forall\alpha.\ T_0;\theta;\delta$. These normalize to $r_1 \sim r_2 : \forall\alpha.\ T_0;\theta;\delta$. If they are blames, $\theta_1(\delta_1(e_1\ T_2))$ and $\theta_2(\delta_2(e_1\ T_2))$ also normalize to blames, and we are done. Let $r_1 = v_1$ and $r_2 = v_2$. Then, by definition, $v_1\ T_1' \simeq v_2\ T_2' : T_0;\theta[\alpha \mapsto R, T_1', T_2'];\delta$ for any R, T_1', T_2'. In particular, $v_1\,\theta_1(\delta_1(T_2)) \simeq v_2\,\theta_2(\delta_2(T_2)) : T_0;\theta[\alpha \mapsto R_{T_2,\theta,\delta}, \theta_1(\delta_1(T_2)), \theta_2(\delta_2(T_2))];\delta$. These normalize to $r_1' \sim r_2' : T_0;\theta[\alpha \mapsto R_{T_2,\theta,\delta}, \theta_1(\delta_1(T_2)), \theta_2(\delta_2(T_2))];\delta$. By Lemma 28, $r_1' \sim r_2' : T_0[T_2/\alpha];\theta;\delta$. By expansion, $\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : T_0[T_2/\alpha];\theta;\delta$.

(T_Cast): We have $e = \langle T_1 \Rightarrow T_2\rangle^l$, $\vdash \Gamma$, $T_1 \parallel T_2$, $\Gamma \vdash T_1$, $\Gamma \vdash T_2$, and $T = T_1 \to T_2$. By the IH, $\Gamma \vdash T_1 \simeq T_1 : *$ and $\Gamma \vdash T_2 \simeq T_2 : *$. By Lemma 33, $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l \simeq \langle T_1 \Rightarrow T_2\rangle^l : T_1 \to T_2$.

(T_Blame): Immediate.

(T_Check): We have $e = \langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l$, $\emptyset \vdash v : T_1$, $\emptyset \vdash e_2 : \mathsf{Bool}$, $\vdash \Gamma$, $\emptyset \vdash \{x{:}T_1 \mid e_1\}$, $e_1[v/x] \longrightarrow^* e_2$, and $T = \{x{:}T_1 \mid e_1\}$. Let $\Gamma \vdash \theta;\delta$. We wish to show that $\theta_1(\delta_1(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) \simeq \theta_2(\delta_2(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) : \{x{:}T_1 \mid e_1\};\theta;\delta$. By the IH, $\theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : \mathsf{Bool};\theta;\delta$ and these normalize to the same result. If the result is false or $\Uparrow l'$ for some l', then $\theta_1(\delta_1(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) \longrightarrow^* \Uparrow l''$ and $\theta_2(\delta_2(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) \longrightarrow^* \Uparrow l''$ for some l'' and we are done. Otherwise, the result is true. Then, by the IH, $v \sim v : T_1;\theta;\delta$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\} \simeq \{x{:}T_1 \mid e_1\} : *$. By definition, $\theta_1(\delta_1(e_1))[v/x] \simeq \theta_2(\delta_2(e_1))[v/x] : \mathsf{Bool};\theta;\delta[v,v/x]$. Then, we have $\theta_1(\delta_1(e_1))[v/x] = e_1[v/x] \longrightarrow^* \mathsf{true}$ and $\theta_2(\delta_2(e_1))[v/x] = e_1[v/x] \longrightarrow^* \mathsf{true}$. By definition, $v \simeq v : \{x{:}T_1 \mid e_1\};\theta;\delta$. By expansion, $\theta_1(\delta_1(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) \simeq \theta_2(\delta_2(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) : \{x{:}T_1 \mid e_1\};\theta;\delta$, as required.

(T_Conv): By Lemma 29.

(T_Exact): We have $e = v$, $\emptyset \vdash v : T$, $\emptyset \vdash \{x{:}T_0 \mid e_0\}$, $e[v/x] \longrightarrow^* \mathsf{true}$, and $T = \{x{:}T_0 \mid e_0\}$. Let $\Gamma \vdash \theta;\delta$. We wish to show that $v \sim v : \{x{:}T_0 \mid e_0\};\theta;\delta$. By the IH, $v \sim v : T_0;\theta;\delta$. Since $\emptyset \vdash \{x{:}T_0 \mid e_0\}$, the only free variable in e_0 is x and $\theta_1(\delta_1(e_0))[v/x] = e_0[v/x] \longrightarrow^* \mathsf{true}$ and $\theta_2(\delta_2(e_0))[v/x] = e_0[v/x] \longrightarrow^* \mathsf{true}$. By definition, $v \sim v : \{x{:}T_0 \mid e_0\};\theta;\delta$.

(T_Forget): By the IH, $\emptyset \vdash v \simeq v : \{x{:}T \mid e\}$, which implies $\Gamma \vdash v \simeq v : T$.

(WF_Base): Trivial.

(WF_TVar): Trivial.

(WF_Fun): By the IH.

(WF_Forall): By the IH.

(WF_Refine): By the IH.

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.

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

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

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

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

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

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

Fig. 6. Subtyping, implication, and closing substitutions

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. This is, at heart, an optimization: since the cast can never fail, there's no point in running it. In this section, we define a subtyping relation for $\mathrm{F_H}$ and prove that upcast elimination is sound. To be clear, the type system of $\mathrm{F_H}$ 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—$\mathrm{F_H}$ doesn't have subsumption!

Our rule for subtyping of refinements differs substantially from λ_{H}'s, mostly because $\mathrm{F_H}$ 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 casts(T) is defined in Figure 4 as the composition of casts necessary to cast from unref(T) to T. So, if application of casts(T) to a value of 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 unref(T_1), then either: (1) v did not pass the checks in casts(T_1), so this value is not in T_1; or (2) v passed the checks in casts(T_1) and $\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\, v$ passed all of the checks in casts(T_2). So, if (1) or (2) hold for all values of type 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 but more dire 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 35. *If $\Gamma, x{:}T_1, \Gamma' \ \vdash\ T_2$ and $\Gamma \vdash e_1 \simeq e_2 : T_1$, then $\Gamma, \Gamma'[e_1/x] \vdash T_2[e_1/x] \simeq T_2[e_2/x] : *$.*

Proof. By induction on $\Gamma, x{:}T_1, \Gamma' \ \vdash\ T_2$.

$\underline{\text{(WF\_Base)}}$: Easy because $T_2 = B = T_2[e_1/x] = T_2[e_1/x]$.

$\underline{\text{(WF\_TVar)}}$: Similar.

$\underline{\text{(WF\_Forall)}}$: We have $T_2 = \forall\alpha.\ T_2'$ and $\Gamma, x{:}T_1, \Gamma', \alpha \ \vdash\ T_2'$. Let $\Gamma, \Gamma'[e_1/x] \vdash \theta; \delta$. We wish to show that $(\forall\alpha.\ T_2')[e_1/x] \simeq (\forall\alpha.\ T_2')[e_2/x] : *; \theta; \delta$. By definition, it suffices to show that for any R, T_1'', T_2'', $T_2'[e_1/x] \simeq T_2'[e_2/x] : *; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$.

Let R, T_1'', T_2'' be given. By the IH, $\Gamma, \Gamma'[e_1/x], \alpha \vdash T_2'[e_1/x] \simeq T_2'[e_2/x] : *$. Since $\Gamma, \Gamma'[e_1/x], \alpha \ \vdash\ \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$, $T_2'[e_1/x] \ \simeq\ T_2'[e_2/x] \ :\ *; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$, as required.

$\underline{\text{(WF\_Fun)}}$: $T_2 = y{:}T_{21} \to T_{22}$ and $\Gamma, x{:}T_1, \Gamma' \ \vdash\ T_{21}$ and $\Gamma, x{:}T_1, \Gamma', y{:}T_{21} \ \vdash\ T_{22}$. Let $\Gamma, \Gamma'[e_1/x] \ \vdash\ \theta; \delta$. We wish to prove $(y{:}T_{21} \to T_{22})[e_1/x] \simeq (y{:}T_{21} \to T_{22})[e_2/x] : *; \theta; \delta$. By definition, it suffices to show $T_{21}[e_1/x] \simeq T_{21}[e_2/x] : *; \theta; \delta$ and for any $v_1 \sim v_2 : T_{21}[e_1/x]; \theta; \delta$, $T_{22}[e_1/x] \simeq T_{22}[e_2/x] : *; \theta; \delta[v_1, v_2/y]$.

By the IH, $\Gamma, \Gamma'[e_1/x] \ \vdash\ T_{21}[e_1/x] \ \simeq\ T_{21}[e_2/x] \ :\ *$, and so $T_{21}[e_1/x] \simeq T_{21}[e_2/x] : *; \theta; \delta$.

By the IH, $\Gamma, \Gamma'[e_1/x], y{:}T_{21}[e_1/x] \vdash T_{22}[e_1/x] \simeq T_{22}[e_2/x] : *$. Let $v_1 \sim v_2 : T_{21}[e_1/x]; \theta; \delta$. Then, $\Gamma, \Gamma'[e_1/x], y{:}T_{21}[e_1/x] \vdash \theta; \delta[v_1, v_2/y]$. Finally, $T_{22}[e_1/x] \simeq T_{22}[e_2/x] : *; \theta; \delta[v_1, v_2/y]$, as required.

(WF_Refine): We have $T_2 = \{y{:}T_2' \mid e_0\}$ and $\Gamma, x{:}T_1, \Gamma' \vdash T_2'$ and $\Gamma, x{:}T_1, \Gamma', y{:}T_2' \vdash e_0 : \mathsf{Bool}$. Let $\Gamma, \Gamma'[e_1/x] \vdash \theta; \delta$. We wish to show that $\{y{:}T_2' \mid e_0\}[e_1/x] \simeq \{y{:}T_2' \mid e_0\}[e_2/x] : *; \theta; \delta$. By definition, it suffices to show that $T_2'[e_1/x] \simeq T_2'[e_2/x] : *; \theta; \delta$ and for any $v_1 \sim v_2 : T_2[e_1/x]; \theta; \delta$, $\theta_1(\delta_1(e_0[e_1/x]))[v_1/y] \simeq \theta_2(\delta_2(e_0[e_2/x]))[v_2/y] : \mathsf{Bool}; \theta; \delta$.

By the IH, $\Gamma, \Gamma'[e_1/x] \vdash T_2'[e_1/x] \simeq T_2'[e_2/x] : *$, and so, $T_2'[e_1/x] \simeq T_2'[e_2/x] : *; \theta; \delta$.

Let $v_1 \sim v_2 : T_2[e_1/x]; \theta; \delta$. By Lemma 26, $v_1 \sim v_2 : T_2; \theta; \delta[\theta_1(\delta_1(e_1)), \theta_2(\delta_2(e_2))/x]$. By assumption, $\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : T_1; \theta; \delta$. So, we have $\Gamma, x{:}T_1, \Gamma', y{:}T_2 \vdash \theta; \delta[\theta_1(\delta_1(e_1)), \theta_2(\delta_2(e_2))/x][v_1, v_2/y]$. Let $\delta' = \delta[\theta_1(\delta_1(e_1)), \theta_2(\delta_2(e_2))/x][v_1, v_2/y]$. Since $\Gamma, x{:}T_1, \Gamma', y{:}T_2 \vdash e_0 \simeq e_0 : \mathsf{Bool}$ by Lemma 34, we have $\theta_1(\delta_1'(e_0)) \simeq \theta_2(\delta_2'(e_0)) : \mathsf{Bool}; \theta; \delta'$. The fact that $\theta_1(\delta_1'(e_0)) = \theta_1(\delta_1(e_0[e_1/x]))[v_1/y]$ and $\theta_2(\delta_2'(e_0)) = \theta_2(\delta_2(e_0[e_2/x]))[v_2/y]$ and Lemma 27 show $\theta_1(\delta_1(e_0[e_1/x]))[v_1/y] \simeq \theta_2(\delta_2(e_0[e_2/x]))[v_2/y] : \mathsf{Bool}; \theta; \delta$ as required.

Lemma 36. *If $T_1 \simeq T_2 : *; \theta; \delta$ and $T_2 \simeq T_1 : *; \theta; \delta$, then $v_1 \sim v_2 : T_1; \theta; \delta$ if and only if $v_1 \sim v_2 : T_2; \theta; \delta$.*

Proof. By induction on the size of T_1 and T_2.
($T_1 = T_2 = B$): Trivial.
($T_1 = T_2 = \alpha$): Trivial.
($T_1 = x{:}T_{11} \to T_{12}$ and $T_2 = x{:}T_{21} \to T_{22}$): By definition, $T_{11} \simeq T_{21} : *; \theta; \delta$ and $T_{21} \simeq T_{11} : *; \theta; \delta$ and $\forall v_1' \sim v_2' : T_{11}; \theta; \delta.\ T_{12} \simeq T_{22} : *; \theta; \delta[v_1', v_2'/x]$ and $\forall v_1' \sim v_2' : T_{21}; \theta; \delta.\ T_{22} \simeq T_{12} : *; \theta; \delta[v_1', v_2'/x]$.

Then,

$$\begin{aligned}
& && v_1 \sim v_2 : T_1; \theta; \delta \\
& \iff && \forall (v_1' \sim v_2' : T_{11}; \theta; \delta). v_1\ v_1' \simeq v_2\ v_2' : T_{12}; \theta; \delta[v_1', v_2'/x] \\
\text{(by the IH)} \quad & \iff && \forall (v_1' \sim v_2' : T_{21}; \theta; \delta). v_1\ v_1' \simeq v_2\ v_2' : T_{22}; \theta; \delta[v_1', v_2'/x] \\
& \iff && v_1 \sim v_2 : T_2; \theta; \delta.
\end{aligned}$$

($T_1 = \forall \alpha.\ T_1'$ and $T_2 = \forall \alpha.\ T_2'$): By definition, $\forall R, T_1'', T_2''. T_1' \simeq T_2' : *; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$ and $\forall R, T_1'', T_2''. T_2' \simeq T_1' : *; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$.

Then,

$$\begin{aligned}
& && v_1 \sim v_2 : T_1; \theta; \delta \\
& \iff && \forall R, T_1'', T_2''. v_1\ T_1'' \simeq v_2\ T_2'' : T_1'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta \\
\text{(by the IH)} \quad & \iff && \forall R, T_1'', T_2''. v_1\ T_1'' \simeq v_2\ T_2'' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta \\
& \iff && v_1 \sim v_2 : T_2; \theta; \delta.
\end{aligned}$$

($T_1 = \{x{:}T_1' \mid e_1\}$ and $T_2 = \{x{:}T_2' \mid e_2\}$): By definition, $T_1' \simeq T_2' : *; \theta; \delta$ and $T_2' \simeq T_1' : *; \theta; \delta$ and

$$\forall (v_1 \sim v_2 : T_1'; \theta; \delta). \theta_1(\delta_1(e_1))[v_1/x] \simeq \theta_2(\delta_2(e_2))[v_2/x] : \mathsf{Bool}; \theta; \delta \qquad (1)$$

and

$$\forall (v_1 \sim v_2 : T_2'; \theta; \delta). \theta_1(\delta_1(e_2))[v_1/x] \simeq \theta_2(\delta_2(e_1))[v_2/x] : \mathsf{Bool}; \theta; \delta. \qquad (2)$$

Then,

$$
\begin{array}{rcl}
 & & v_1 \sim v_2 : \{x{:}T_1' \mid e_1\};\theta;\delta \\
 & \iff & \left\{\begin{array}{l} v_1 \sim v_2 : T_1';\theta;\delta \\ \theta_1(\delta_1(e_1))[v_1/x] \longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e_1))[v_2/x] \longrightarrow^* \mathsf{true} \end{array}\right. \\
\left(\begin{array}{l} \text{by the IH} \\ \text{by (1)} \\ \text{by (2)} \end{array}\right) & \iff & \left\{\begin{array}{l} v_1 \sim v_2 : T_2';\theta;\delta \\ \theta_2(\delta_2(e_2))[v_2/x] \longrightarrow^* \mathsf{true} \\ \theta_1(\delta_1(e_2))[v_1/x] \longrightarrow^* \mathsf{true} \end{array}\right. \\
 & \iff & v_1 \sim v_2 : \{x{:}T_2' \mid e_2\};\theta;\delta.
\end{array}
$$

Lemma 37 (Upcast lemma). *If $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_1$ and $\Gamma \vdash T_2$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l \simeq \lambda x{:}T_1.\ x : T_1 \to T_2$.*

Proof. By induction on the subtyping derivation.
(S_Base): Easy. We have $T_1 = B$ and $T_2 = B$. Let $\Gamma \vdash \theta;\delta$. We wish to show that $\theta_1(\delta_1(\langle B \Rightarrow B\rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : B \to B;\theta;\delta$. Let $v_1 \sim v_2 : B;\theta;\delta$. We must show that $\theta_1(\delta_1(\langle B \Rightarrow B\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : B;\theta;\delta[v_1,v_2/z]$ for fresh z, but these normalize to $v_1 \sim v_2 : B;\theta;\delta[v_1,v_2/z]$. Lemma 27 finishes the case.
(S_TVar): Similar to the case for S_Base.
(S_Fun): We have $T_1 = y{:}T_{11} \to T_{12}$ and $T_2 = y{:}T_{21} \to T_{22}$ and $\Gamma \vdash T_{21} <: T_{11}$ and $\Gamma, y{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] <: T_{22}$.

Let $\Gamma \vdash \theta;\delta$. We wish to show that $\theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22}\rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : (y{:}T_{11} \to T_{12}) \to (y{:}T_{21} \to T_{22});\theta;\delta$. Let $v_1 \sim v_2 : y{:}T_{11} \to T_{12};\theta;\delta$. We mush show that $\theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22}\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : (y{:}T_{21} \to T_{22});\theta;\delta[v_1,v_2/z]$ for fresh z.

If $T_1 = T_2$, then we have $\theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22}\rangle^l))\, v_1 \longrightarrow v_1$ and $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 \longrightarrow v_2$ and, by definition, we are done.

Otherwise, $\theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22}\rangle^l))\, v_1 \longrightarrow \theta_1(\delta_1(\lambda y{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{21}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, y))))$ and $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 \longrightarrow v_2$. So, it suffices to show that $\theta_1(\delta_1(\lambda y{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{21}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, y)))) \sim v_2 : (y{:}T_{21} \to T_{22});\theta;\delta[v_1,v_2/z]$. Let $v_1' \sim v_2' : T_{21};\theta;\delta[v_1,v_2/z]$. We will show that $\theta_1(\delta_1(\lambda y{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{21}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, y))))\, v_1' \simeq v_2\, v_2' : T_{22};\theta;\delta[v_1,v_2/z][v_1',v_2'/y]$.

By the IH, $\Gamma \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l \simeq \lambda x{:}T_{21}.\ x : T_{21} \to T_{11}$. So, $\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l))\, v_1' \simeq (\lambda x{:}\theta_1(\delta_1(T_{21})).\ x)\, v_2' : T_{11};\theta;\delta[v_1',v_2'/z']$ for fresh z''. Then, for some v_1'', $\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l))\, v_1' \longrightarrow v_1''$ and $v_1'' \simeq v_2' : T_{11};\theta;\delta[v_1',v_2'/z']$.

By assumption, $v_1\, v_1'' \simeq v_2\, v_2' : T_{12};\theta;\delta[v_1'',v_2'/y]$. These normalize to $v_1''' \simeq v_2''' : T_{12};\theta;\delta[v_1'',v_2'/y]$.

Now, we can show that $\Gamma, y{:}T_{21} \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l\, y \simeq y : T_{11}$,

$\Gamma \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l \simeq \lambda x{:}T_{21}.\ x : T_{21} \to T_{11}$, and use Lemma 35 to obtain $\Gamma, y{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \simeq T_{12} : *$ (note that $T_{12} = T_{12}[y/y]$). Then, by Lemma 36 (note that it is easy to show that $\Gamma \vdash T_1 \simeq T_2 : *$ if and only if $\Gamma \vdash T_2 \simeq T_1 : *$), we have $v_1''' \simeq v_2''' : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y];\theta;\delta[v_1'',v_2'/y]$.

On the other hand, by the other IH, $\Gamma, y{:}T_{21} \vdash \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{22}\rangle^l \simeq \lambda x{:}(T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y]).\ x : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \to T_{22}$. Since

$\Gamma, y{:}T_{21} \vdash \theta; \delta[v_1', v_2'/y]$, $\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l)) \sim \theta_2(\delta_2(\lambda x{:}T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2'/y].\, x)) : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \rightarrow T_{22}; \theta; \delta[v_1', v_2'/y]$. So, $\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l))\, v_1''' \simeq \theta_2(\delta_2((\lambda x{:}T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2'/y].\ x)))\, v_2''' : T_{22}; \theta; \delta[v_1', v_2'/y][v_1''', v_2'''/z''']$ for fresh z'''. They normalize to $v_1'''' \sim v_2''' : T_{22}; \theta; \delta[v_1', v_2'/y][v_1''', v_2'''/z''']$.

Now, we have

$$\begin{array}{ll}
& \theta_1(\delta_1(\lambda y{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{22}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, y)))))\, v_1' \\
\longrightarrow & \theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1')))) \\
\longrightarrow^* & \theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l))\, (v_1\, v_1'') \\
\longrightarrow^* & \theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l))\, v_1''' \\
\longrightarrow^* & v_1''''
\end{array}$$

and $v_2\, v_2' \longrightarrow^* v_2'''$ and $v_1'''' \sim v_2''' : T_{22}; \theta; \delta[v_1', v_2'/y][v_1''', v_2'''/z''']$. By expansion, we have what we wanted to show.

(S_Forall): We have $T_1 = \forall\alpha.\ T_1'$ and $T_2 = \forall\alpha.\ T_2'$ and $\Gamma, \alpha \vdash T_1' <: T_2'$.

Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : T_1 \rightarrow T_2; \theta; \delta$. Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : T_2; \theta; \delta[v_1, v_2/z]$ for fresh z.

If $T_1 = T_2$, then we have $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \longrightarrow v_1$ and $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 \longrightarrow v_2$ and we are done.

Otherwise, $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \longrightarrow \Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, (v_1\, \alpha))$. So, it suffices to show that $\Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, (v_1\, \alpha)) \simeq v_2 : T_2; \theta; \delta[v_1, v_2/z]$. Let R, T_1'', T_2'' be given. We must show that $\Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, (v_1\, \alpha))\ T_1'' \simeq v_2\ T_2'' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1, v_2/z]$. Since $\Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, (v_1\, \alpha))\ T_1'' \longrightarrow \theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha]\rangle^l))\, (v_1\ T_1'')$, it suffices to show that $\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha]\rangle^l))\, (v_1\ T_1'') \simeq v_2\ T_2'' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1, v_2/z]$.

By assumption $v_1\ T_1'' \simeq v_2\ T_2'' : T_1; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$. These normalize to $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$. By the IH, $\Gamma, \alpha \vdash \langle T_1' \Rightarrow T_2'\rangle^l \simeq \lambda x{:}T_1'.\ x : T_1' \rightarrow T_2'$ and so, $\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha]\rangle^l)) \sim \lambda x{:}T_1'[T_2''/\alpha].\ x : T_1' \rightarrow T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$. Then, by definition, $\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha]\rangle^l))\, v_1' \simeq (\lambda x{:}T_1'[T_2''/\alpha].\ x)\, v_2' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1', v_2'/z']$ for fresh z'. They normalize to $v_1'' \sim v_2' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1', v_2'/z']$.

By expansion, $\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha]\rangle^l))\, (v_1\ T_1'') \simeq v_2\ T_2'' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1, v_2/z]$, as required.

(S_Refine): We have $\Gamma \vdash \mathrm{unref}(T_1) <: \mathrm{unref}(T_2)$ and $\Gamma, x{:}\mathrm{unref}(T_1) \vdash \mathrm{casts}(T_1)\, x \supset \mathrm{casts}(T_2)\, (\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\, x)$. Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : T_1 \rightarrow T_2; \theta; \delta$. Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : T_2; \theta; \delta[v_1, v_2/z]$ for fresh z.

We have three cases according to how the LHS reduces.

($T_2 = \mathrm{unref}_1^j(T_1)$ for some j): Then, we have

$$\begin{array}{lll}
& & \theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \\
\text{(by E\_Forget)} & \longrightarrow^* & \theta_1(\delta_1(\langle T_2 \Rightarrow T_2\rangle^l))\, v_1 \\
\text{(by E\_Refl)} & \longrightarrow & v_1.
\end{array}$$

Since $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\,v_2 \longrightarrow v_2$, we have $v_1 \sim v_2 : T_1;\theta;\delta[v_1,v_2/z]$. Since $\mathrm{unref}_1^j(T_1) = T_2$, we also have $v_1 \sim v_2 : T_2;\theta;\delta[v_1,v_2/z]$. So, by expansion, $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\,v_2 : T_2;\theta;\delta[v_1,v_2/z]$, as required.
($T_1' = \mathrm{unref}_1^j(T_1)$ for some j and $T_2 = \{x{:}T_1' \mid e_2\}$): Then, we have

$$\begin{array}{lll}
 & & \theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,v_1 \\
\text{(by E\_Forget)} & \longrightarrow^* & \theta_1(\delta_1(\langle T_1' \Rightarrow \{x{:}T_1' \mid e_2\}\rangle^l))\,v_1 \\
\text{(by E\_Check)} & \longrightarrow & \langle \theta_1(\delta_1(\{x{:}T_1' \mid e_2\})), \theta_1(\delta_1(e_2))[v_1/x], v_1\rangle^l.
\end{array}$$

By assumption, $v_1 \simeq v_2 : \mathrm{unref}(T_1);\theta;\delta$ and $\theta_1(\delta_1(\mathrm{casts}(T_1)))\,v_1 \longrightarrow^* v_1$. By Imp, $\theta_1(\delta_1(\mathrm{casts}(T_2)))\,(\theta_1(\delta_1(\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l))\,v_1) \longrightarrow^* v_1'$ for some v_1'. By inspecting the reduction sequence, we have

$$\begin{array}{lll}
 & & \mathrm{casts}(T_2)\,(\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\,v_1) \\
\text{(by E\_Refl)} & \longrightarrow & \mathrm{casts}(T_2)\,v_1 \\
 & \longrightarrow^* & \langle T_1' \Rightarrow \{x{:}T_1' \mid e_2\}\rangle^l\,v_1 \\
\text{(by E\_Check)} & \longrightarrow & \langle \theta_1(\delta_1(\{x{:}T_1' \mid e_2\})), \theta_1(\delta_1(e_2))[v_1/x], v_1\rangle^l \\
 & \longrightarrow^* & \langle \theta_1(\delta_1(\{x{:}T_1' \mid e_2\})), \mathsf{true}, v_1\rangle^l \\
 & \longrightarrow & v_1.
\end{array}$$

Since $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\,v_2 \longrightarrow v_2$, we have $v_1 \sim v_2 : T_1;\theta;\delta[v_1,v_2/z]$. Since $\mathrm{unref}_1^j(T_1) = T_1'$, we also have $v_1 \sim v_2 : T_1';\theta;\delta[v_1,v_2/z]$. Finally, $\theta_1(\delta_1(e_2))[v_1/x] \longrightarrow^*$ true gives $v_1 \sim v_2 : \{x{:}T_1' \mid e_2\};\theta;\delta[v_1,v_2/z]$. By expansion, $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\,v_2 : T_2;\theta;\delta[v_1,v_2/z]$, as required.
(Otherwise): We have

$$\begin{array}{lll}
 & & \theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,v_1 \\
\text{(by E\_Forget)} & \longrightarrow^* & \theta_1(\delta_1(\langle \mathrm{unref}(T_1) \Rightarrow T_2\rangle^l))\,v_1 \\
\text{(by E\_Precheck)} & \longrightarrow^* & e
\end{array}$$

where $e = \theta_1(\delta_1(\langle \mathrm{unref}_1(T_2) \Rightarrow T_2\rangle^l\,(\langle \mathrm{unref}_2(T_2) \Rightarrow \mathrm{unref}_1(T_2)\rangle^l\,(\cdots\,(\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\,v_1)))))$.
By the IH, $\Gamma \vdash \langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l \simeq \lambda x{:}\mathrm{unref}(T_1).\ x : (\mathrm{unref}(T_1) \to \mathrm{unref}(T_2))$. So, $\theta_1(\delta_1(\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l))\,v_1 \simeq \theta_1(\delta_1(\lambda x{:}\mathrm{unref}(T_1).\ x))\,v_2 : \mathrm{unref}(T_2);\theta;\delta[v_1,v_2/z]$. They normalize to $v_1' \sim v_2 : \mathrm{unref}(T_2);\theta;\delta[v_1,v_2/z]$.
By assumption, $v_1 \simeq v_2 : \mathrm{unref}(T_1);\theta;\delta$ and $\theta_1(\delta_1(\mathrm{casts}(T_1)))\,v_1 \longrightarrow^* v_1$. By Imp, $\theta_1(\delta_1(\mathrm{casts}(T_2)\,((\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l)\,v_1))) \longrightarrow \theta_1(\delta_1(\mathrm{casts}(T_2)))\,v_1' \longrightarrow^*$ v_1'' for some v_1''. Inspecting this reduction sequence gives $v_1'' = v_1'$.
By Lemma 34, $\Gamma \vdash T_2 \simeq T_2 : *$, which shows $\theta_2(\delta_2(\mathrm{casts}(T_2)))\,v_2 \longrightarrow^* v_2'$ for some v_2'. So, $v_1' \sim v_2 : T_2;\theta;\delta[v_1,v_2/z]$. Finally, by expansion, $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\,v_2 : T_2;\theta;\delta[v_1,v_2/z]$, as required.

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 $\mathrm{F_H}$ does; we then discuss how $\mathrm{F_H}$ 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. [3] 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. [2] 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., but we leave it 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]

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

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

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 $\emptyset \vdash \mathrm{ty}\,(k) <: \{x{:}B \mid e\}$—are used to show preservation of active checks. (One such assignment is $\mathrm{ty}\,(k) = \{x{:}B \mid x = k\}$.) The two systems use subtyping to the 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 with a runtime rule that allows us to type values with any refinement they satisfy:

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

This eliminates one use of subtyping and also 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.

There is another, unintended consequence of including subtyping in the type system itself. KF's cast insertion algorithm uses a three-valued theorem prover to compile source-level programs into λ_H terms with casts: if the prover says "yes", then subtyping holds and no cast is necessary; if the prover says "maybe", then subtyping may or may not hold, so a cast is inserted; if the prover says "no", then the program is rejected. Rejecting such programs is tempting, but we feel it is too conservative: a cast that isn't an upcast doesn't *always* fail, it just *may* fail. A program should be rejected not when a non-upcast is needed, but when attempting to cast between types that don't share any values.

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. Two other standard extensions are natural next steps: general recursion and state. With the introduction of abstract types, there is room to draw connections between the client/server blame from Section 2 and Findler and Felleisen-style client/server blame.

Finally, we are curious to see what we can do in a contract language with the reasoning principles derivable from relational parametricity.

Acknowledgments

Stephanie Weirich provided many insights throughout. 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., Siek, J., Wadler, P.: Blame for all. In: Principles of Programming Languages (POPL) (2011)
3. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Workshop on Script-to-Program Evolution (STOP) (2009)
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)