Polymorphic Manifest Contracts, Revised and Resolved

TARO SEKIYAMA and ATSUSHI IGARASHI, Kyoto University
MICHAEL GREENBERG, Princeton University

Manifest contracts track precise program properties by refining types with predicates—e.g., $\{x{:}\mathsf{Int} \mid x > 0\}$ denotes the positive integers. Contracts and *polymorphism* make a natural combination: programmers can give strong contracts to abstract types, precisely stating pre- and post-conditions while hiding implementation details—for example, an abstract type of stacks might specify that the pop operation has input type $\{x{:}\alpha\ \mathsf{Stack} \mid \mathsf{not}\,(\mathsf{empty}\,x)\}$.

Belo et al. [2011] defined $\mathrm{F_H}$, a polymorphic calculus with manifest contracts and dependent functions, and established fundamental properties including type soundness and relational parametricity. Greenberg [2013] fixed some but not all of the metatheoretical problems in $\mathrm{F_H}$'s type conversion relation. We define $\mathrm{F}^{\sigma}_{\mathrm{H}}$, which resolves the issues in both prior versions of $\mathrm{F_H}$'s with new semantics for substitution and a new conversion relation.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.2.4 [**Software engineering**]: Software/Program Verification—*Programming by contract*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Design, Theory

Additional Key Words and Phrases: contracts, refinement types, preconditions, postconditions, dynamic checking, runtime verification, parametric polymorphism, abstract datatypes, syntactic proof, logical relations

1. INTRODUCTION

Software contracts allow programmers to state precise properties as concrete predicates written in the same language as the rest of the program; for example, contracts can indicate that a function takes a non-empty list to a positive integer. 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 [2002] introduced "higher-order contracts" for functional languages, defining the first runtime verification semantics for a functional language; these contracts can take one of two forms: predicate contracts given by a Boolean function and function contracts $c_1 \mapsto c_2$, which designate contracts for a function's input and output by c_1 and c_2, respectively. Greenberg, Pierce, and Weirich [2010] contrast two different approaches to contracts according to how contracts and types interact with each other: in the latent approach, contracts and types live in different worlds (indeed, there may be no types at all, as in Racket's contract system [Flatt and PLT 2010; PLT 2014]); in the manifest approach, contracts are types–the type system itself makes contracts 'manifest'–and dynamic

This article extends, clarifies, and revises Belo et al. [2011] and Chapter 3 of Greenberg [2013]. It is *not* merely an extended version with proofs. 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.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

contract checking is expressed by type conversions, which are more commonly called casts.

Manifest contracts are a sensible choice for combining contracts and other type-based abstraction mechanisms, such as, abstract datatypes (ADTs). Abstract datatypes already use the type system to mediate access to abstractions; manifest contracts allow types to exercise a still finer grained control. To motivate the combination of contracts and ADTs, consider the interface of an abstract datatype (ADT) modeling the natural numbers, written in an ML-like language:

```
module type NAT =
sig
   type t
   val zero : t
   val succ : t -> t
   val isZ  : t -> bool
   val pred : t -> t
end
```

It is an *abstract* datatype because the actual representation of `t` is hidden: users of `NAT` interact with it through the constructors and operations provided. The `zero` constructor represents 0; the `succ` constructor takes a natural and produces its successor. The predicate `isZ` determines whether a given natural is zero. The `pred` operation takes a natural number and returns its predecessor.

This interface, however, is not fine-grained enough to prevent misuse of partial operations. For example, `pred` can be applied to `zero`, whereas the mathematical natural-number predecessor operation isn't defined for zero.

Using contracts, we can explicitly specify the constraint that an argument to `pred` is not zero:

```
module type NAT =
sig
   type t
   val zero : t
   val succ : t -> t
   val isZ  : t -> bool
   val pred : {x:t | not (isZ x)} -> t
end
```

The type `{x:t | not (isZ x)}` is a refinement type and denotes the set of values `x` such that `not (isZ x)` evaluates to true. So, this new interface does not allow `pred` to be applied to zero.

In this article, we study the interaction between type abstraction and (manifest) contracts. We introduce a polymorphic manifest contract calculus F_H^σ, which is an extension of System F with manifest contracts, and investigate its properties. Actually, F_H^σ is based on another polymorphic manifest contract calculus F_H proposed by Belo et al. [2011] and fixes a few technical flaws (which we will discuss later) found in F_H.

Both F_H and F_H^σ scale up to polymorphism by diverging from earlier manifest calculi such as λ_H [Flanagan 2006], a simply typed manifest contract calculus—for two reasons.

First, we would naturally need so-called "general refinements", where the underlying type `T` in a refinement type `{x:T|e}` can be an arbitrary type (not only base types like `bool` and `int` but also function and forall types). To see such a need, let's look again at our NAT abstract datatype. If we implement the abstract type t by the Church encoding, we have to implement pred as a function of type

```
{x:∀a.a->(a->a)->a | not (isZ x)} -> ∀a.a->(a->a)->a
```

in which the Church natural number type ∀a.a->(a->a)->a is substituted for t. More generally, if we can put contracts on type variables, then the contracts on our ADTs could need to allow refinements on function, forall, and even refinement types. These "general refinements" aren't possible in existing manifest calculi, which restrict refinements to base types.

Second, the metatheory for existing calculi has been too complicated to extend easily [Greenberg et al. 2010; Knowles and Flanagan 2010]. The complexity is caused by denotational technique introduced to prove semantic type soundness. We explain why they need it in detail when we discuss related work (Section 8), but, briefly, the problem is subtyping. Subtyping is crucial to prove type soundness in their approaches. In both Greenberg et al. and Knowles and Flanagan, subtyping between refinement types is what ultimately requires semantic typing to avoid a dangerous circularity in the mutually recursive definition of the typing and subtyping judgments. The denotations of types used are, however, harder to scale than standard syntactic methods (i.e., progress and preservation).

Our earlier manifest calculus F_H addresses these two issues: first, the calculus allows general refinements; second, it replaces subtyping with a syntactic conversion relation, which allows for a simpler, more scalable, syntactic metatheory. Furthermore, Belo et al. showed that eliminating subtyping doesn't lose useful reasoning principles: in Section 5 of their paper, Belo et al. [2011] define subtyping *post facto* and recover an "upcast" lemma from Knowles and Flanagan [Knowles and Flanagan 2010] showing that any value of a type can be treated as its supertypes.

Unfortunately, however, a few technical flaws have been found in the metatheory of F_H. A first issue is that a key lemma about the conversion relation is false. Greenberg [2013] resolved this problem by changing the conversion relation to the one based on what we call *common subexpression reduction*. The more challenging one is that the proofs of type soundness and parametricity of F_H rests on a *wrong* conjecture, called *cotermination*, which says that reduction of subterms does not affect the evaluation result of the whole refinement term.[1] We examine these problems in Section 7 in detail; briefly speaking, the problem is in the operational semantics where the substitution can change which reduction rules are chosen at runtime.

In this article, we introduce a new calculus F_H^σ that resolves the technical flaws in F_H. We call our calculus F_H^σ because it takes the F_H from Belo et al. [2011] and Greenberg [2013] and introduces a new substitution semantics using *delayed substitutions*, which we write σ. Thanks to delayed substitution, the semantics of F_H^σ can choose reduction rules independently of substitution; this property is crucial when we prove cotermination. We can finally show that type soundness, parametricity, and cotermination all hold in F_H^σ—*without leaving any conjectures*.

1.1. Contributions and Outline

After giving an overview of our calculus in Section 2, we define F_H^σ in Section 3. In Section 4, we develop longer and more detailed examples than exist in the literature so far. We prove type soundness in Section 5, fixing Belo et al. [2011] with common-subexpression reduction from Greenberg [2013] and our novel delayed substitutions.

[1]In the end of Section 4 of Belo et al. [2011], the authors write "our proof of type soundness in Section 3 relies on much simpler properties of parallel reduction, which we *have* proved." as if the type soundness proof didn't depend on cotermination, but this claim also turns out to be false.

We prove parametricity in Section 6; along with the proofs of cotermination and type soundness in the prior section, this constitutes the first conjecture-free metatheory for the combination of System F and manifest contracts, resolving issues in prior versions of $\mathrm{F_H}$. Section 7 compares $\mathrm{F}_\mathrm{H}^\sigma$ with two variants of polymorphic manifest contracts [Belo et al. 2011; Greenberg 2013]. Finally, we discuss broader related work in Section 8, concluding in Section 9.

2. TECHNICAL OVERVIEW

Starting with a review of basics of casts, which are a mechanism to enforce contracts in manifest calculi, we informally discuss issues in introducing parametric polymorphism.

2.1. Basics of casts

Like other manifest calculi, $\mathrm{F}_\mathrm{H}^\sigma$ uses casts to dynamically enforce contracts. Ordinarily, a cast is just written $\langle T_1 \Rightarrow T_2\rangle^l$; we call T_1 the source type and T_2 the target type. The l superscript is a *blame label*, an abstract source location used to differentiate between different casts and identify the source of failures. These failures are indicated by *blame*, an uncatchable exception with a blame label attached; we write this exception $\Uparrow l$ and pronounce it “blame l”. To use a cast, one applies it—like a function—to a value v with the source type T_1. Running the cast produces a similar value of the target type T_2 if there’s no problem treating v as a T_2. If there is a problem, then the cast will “raise” blame at its label, terminating the program.

Early work on contracts built on simple types as a framework, using a language of types comprising refinements of base types $\{x{:}B \mid e\}$ and dependent functions $x{:}T_1 \to T_2$ [Flanagan 2006; Wadler and Findler 2009; Greenberg et al. 2010].

At base types, casts do one of two things: they either return the value they were applied to, or “raise blame”. For example, consider a cast from integers Int to positive integers, $\{x{:}\mathsf{Int} \mid x > 0\}$. We write this cast $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l$, picking an arbitrary label l. If we apply this cast to 5, we expect to get 5 back, since $5 > 0$. That is,

$$\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\, 5 \longrightarrow^* 5.$$

On the other hand, suppose we apply the same cast to 0. This cast fails, since 0 is certainly not greater than itself. When the cast fails, it will raise blame with its label:

$$\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\, 0 \longrightarrow^* \Uparrow l.$$

When checking predicate contracts, only the target type matters—the type system will guarantee that whatever value we have is well typed at the source type, i.e., satisfies any predicates it has. For example, we’ll always have

$$\langle \{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int}\rangle^l\, v \longrightarrow v$$

immediately, for all values v. The application will only be well typed if v actually *is* a positive number, but, operationally, the source type doesn’t matter in this case.

Unlike casts between base types and refinements, casts between function types aren’t checked immediately. Instead, casts at function types wrap their argument up, decomposing the function cast into two parts: one cast for the domain and one for the codomain. In this way, checking is deferred until the cast function is called. Suppose we have a function f of type $\mathsf{Int} \to \mathsf{Int}$ and we want to ensure that maps positives to numbers greater than 5, casting it to $\{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}$. The cast decomposes as follows:

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

Both casts in the wrapper have the same blame label as the original cast. Notice that the domains of the function types are treated contravariantly; note the inner term in the wrapped term: $f\,(\langle\{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int}\rangle^{l}\, x)$. In this case, the domain cast $\langle\{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int}\rangle^{l}\, x$ will never fail: every positive integer is also an integer. The codomain cast is covariant: $\langle \mathsf{Int} \Rightarrow \{y{:}\mathsf{Int} \mid y > 5\}\rangle^{l}$ checks that f returns a number greater than 5. Since not every integer is greater than 5, this cast will fail if f returns a number less than or equal to 5.

Let's consider a few concrete choices of the function f. First, we cast the identity function to $\{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}$. We can safely apply the cast function to 6:

$$\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}\rangle^{l}\,(\lambda x{:}\mathsf{Int}.\ x)\, 6 \longrightarrow^{*} 6.$$

In general, the identity function does not take positives to numbers greater than 5. But when we apply the cast function to 6, which happens to satisfy the codomain contract, no blame is raised. However, when we apply the cast identity function to a value that doesn't satisfy the codomain contract, say 2, blame will be raised:

$$\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}\rangle^{l}\,(\lambda x{:}\mathsf{Int}.\ x)\, 2 \longrightarrow^{*} \Uparrow l.$$

Contrast this with the sorts of static checks offered by type systems: contract systems raise blame only when a violation is *detected*; type systems are usually conservative, signaling errors when a violation is *possible*.

Some functions will *never* work. If we cast the constant zero function to $\{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}$, it will raise blame for any value:

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

The constant zero function, which *never* returns a value satisfying the codomain contract $\{y{:}\mathsf{Int} \mid y > 5\}$. No matter what value we apply the cast function to, it will always raise blame. Cast at function types are deferred, though: if we never call the cast function, it never has the opportunity to raise blame. Since casts check contracts dynamically, they only detect errors in parts of the program that are explored at runtime.

$\mathrm{F}_{\mathrm{H}}^{\sigma}$'s type system rules out directly applying a function with domain type $\{x{:}\mathsf{Int} \mid x > 0\}$ to 0. It is an important property of $\mathrm{F}_{\mathrm{H}}^{\sigma}$ that 0 doesn't have type $\{x{:}\mathsf{Int} \mid x > 0\}$! One can try to cast 0 from Int to $\{x{:}\mathsf{Int} \mid x > 0\}$, but this will always fail:

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

Finally, $\mathrm{F}_{\mathrm{H}}^{\sigma}$ supports dependent function types. For example, the type $x{:}\mathsf{Int} \to \{y{:}\mathsf{Int} \mid y > x\}$ is inhabited by functions over integers which produce results greater than their inputs. Dependent functions allow for very precise specifications. For example, $x{:}\mathsf{Float} \to \{y{:}\mathsf{Float} \mid \left|y^2 - x\right| < \epsilon\}$ specifies the square-root function. The exact unwinding rule for dependent functions is slightly subtle—see the discussion of E_FUN in Section 3.

2.2. Extending manifest contracts to polymorphism

Polymorphism is a cornerstone of functional programming. First, polymorphism allow programmers to reuse higher-order functions like `compose`:

$$\begin{array}{lcl} \mathsf{compose} & : & \forall\alpha.\forall\beta.\forall\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma) \\ \mathsf{compose} & = & \Lambda\alpha.\ \Lambda\beta.\ \Lambda\gamma.\ \lambda f{:}(\alpha \to \beta).\ \lambda g{:}(\beta \to \gamma).\ \lambda x{:}\alpha.\ g\,(f\,x) \end{array}$$

Perhaps more importantly, polymorphism is critical for defining abstract datatypes and expressing modularity. For example, the standard encoding of products and sums are as follows:

$$\begin{array}{rcl} T_1 \times T_2 & = & \forall\alpha.(T_1 \to T_2 \to \alpha) \to \alpha \\ (e_1, e_2)_{T_1 \times T_2} & = & \Lambda\alpha.\ \lambda f{:}(T_1 \to T_2 \to \alpha).\ f\ e_1\ e_2 \\ \\ T_1 + T_2 & = & \forall\alpha.(T_1 \to \alpha) \to (T_2 \to \alpha) \to \alpha \\ (\mathsf{L}\ e)_{T_1 + T_2} & = & \Lambda\alpha.\ \lambda f_1{:}(T_1 \to \alpha).\ \lambda f_2{:}(T_2 \to \alpha).\ f_1\ e \\ (\mathsf{R}\ e)_{T_1 + T_2} & = & \Lambda\alpha.\ \lambda f_1{:}(T_1 \to \alpha).\ \lambda f_2{:}(T_2 \to \alpha).\ f_2\ e \end{array}$$

These encodings mean that calculi with polymorphism indirectly model calculi with data structures. Encodings of existentials allow for truly 'abstract' datatypes along with simple models of module systems.

Adding polymorphism is a vital step in scaling up the theory of manifest contracts: few functional programmers would want to use a language without polymorphism in some form. But that's not the only reason to add polymorphism: manifest contracts add expressivity to polymorphism. We discuss this more below in Section 4, but as a preview: manifest contracts allow us to express *dependent* pairs, e.g., lists paired with their lengths, public keys paired with their constituent primes, etc.

Unfortunately, adding polymorphism to manifest contracts isn't as simple as just adding some extra syntax terms. The crux of the matter is this: we need to be able to write $\{x{:}\alpha \mid e\}$ for our refinements to interact with abstract datatypes in a useful way. What types can be instantiated for the type variable α?

Earlier work restricts refinements to base types, forbidding refinements like $\{f{:}(\mathsf{Int} \to \mathsf{Int}) \mid f\ 0 = 0\}$. Why? The core issue that led to forbidding refinements of function types was subject reduction. The cast $\langle\{x{:}T \mid e_1\} \Rightarrow \{x{:}T \mid e_2\}\rangle^l\ v$ will reduce to either v or $\Uparrow l$. To have subject reduction, we need to be able to give the value v the type $\{x{:}T \mid e_1\}$ and $\{x{:}T \mid e_2\}$. When T is restricted to base types, we can assign constants *most specific types*, e.g., 2 has type $\{x{:}\mathsf{Int} \mid x = 2\}$. Constants are assigned types that are subtypes of any refinement they pass checks for.

These 'selfified' types [Ou et al. 2004] work fine for constants, but what should we do for functions? It may be that we could find an ad hoc solution for refinements of functions. But to truly have polymorphism, we must be able to instantiate type variables α with *any* type, even a type that's already refined, like $\{x{:}\mathsf{Int} \mid x \geq 0\}$. We call this *general refinement*. If we instantiate a refined type variable with an already refined type, what happens? How high can the stack of refinements go?

$\mathrm{F}_{\mathrm{H}}^{\sigma}$ uses a new metatheory that does away with subtyping entirely, allowing general refinement. The crucial changes are a rule that says values can be typed at any refinement they satisfy (T_EXACT in Figure 3) and a type conversion relation replacing subtyping ($\equiv$ in Figure 4).

2.3. Delayed substitution semantics

Introducing general refinements means defining a new semantics for casts: what can be cast to what, and how do casts evaluate? The decisions here are critical. Type variables can only be cast to themselves, just like base variables. Casts between refinements must, at their core, be between similar base types, type variables, or functions. A cast $\langle T_1 \Rightarrow T_2\rangle^l$ evaluates in several steps, described in detail below (Section 3). In general, the semantics first forgets all of the refinements on T_1 and then starts checking the refinements on T_2 from the inside out. Refinement checking bottoms out in a reflexivity rule: casts of the form $\langle T \Rightarrow T\rangle^l$ just disappear (E_REFL in Figure 2).

Unlike earlier manifest calculi [Flanagan 2006; Wadler and Findler 2009; Greenberg et al. 2010], in which a reflexive cast disappears only when the target (and also

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\textbf{Terms, substitutions, and contexts}} \\
\mathsf{Ty} \ni T & ::= & B \mid \alpha \mid x{:}T_1 \to T_2 \mid \forall\alpha.T \mid \{x{:}T \mid e\} \\
\sigma & \in & (\mathsf{TmVar} \overset{\text{fin}}{\rightharpoonup} \mathsf{Tm}) \times (\mathsf{TyVar} \overset{\text{fin}}{\rightharpoonup} \mathsf{Ty}) \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T \mid \Gamma, \alpha \\
\multicolumn{3}{l}{\textbf{Terms, values, results, and evaluation contexts}} \\
\mathsf{Tm} \ni e & ::= & x \mid k \mid \mathrm{op}\,(e_1, \ldots, e_n) \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid e_1\ e_2 \mid e\ T \mid \\
& & \langle T_1 \Rightarrow T_2\rangle^l_\sigma \mid \Uparrow l \mid \langle \{x{:}T \mid e_1\}, e_2, v\rangle^l \\
v & ::= & k \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid \langle T_1 \Rightarrow T_2\rangle^l_\sigma \\
r & ::= & v \mid \Uparrow l \\
E & ::= & [\,]\ e_2 \mid v_1\ [\,] \mid [\,]\ T \mid \langle \{x{:}T \mid e\}, [\,], v\rangle^l \mid \mathrm{op}(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)
\end{array}
$$

Fig. 1. Syntax for $\mathrm{F}^\sigma_\mathrm{H}$

source) type is a base type[2], this reflexivity rule works for *any* type T. This is motivated by parametricity—$\langle \alpha \Rightarrow \alpha\rangle^l$ should behave the same whatever the type variable α is bound to and the only reasonable behavior is to disappear like the identity function.

The reflexivity rule, however, led to a problem in earlier formulations of manifest contracts with polymorphism—Belo et al. [2011] and Greenberg [2013]. We discuss the problem at length in Section 7, but essence is that substitution could interfere with casts, causing certain terms to incorrectly use reflexivity when they should have used a different rule. The consequences of this interference were metatheoretically dire. While we haven't been able to produce well typed terms with unsound behavior, the prior type soundness proofs are not correct.

$\mathrm{F}^\sigma_\mathrm{H}$ uses *delayed substitutions* σ to ensure that substitution doesn't interfere with how casts evaluate. A delayed substitution is just a map from variables to terms and types. The reason they're 'delayed' is that casts ignore substitutions when deciding what steps to take to check values. Applying a *cast* $\langle T_1 \Rightarrow T_2\rangle^l_\sigma$ with delayed substitution σ to a value of type $\sigma(T_1)$ will ensure that the value behaves like a $\sigma(T_2)$.

3. DEFINING $\mathrm{F}^\sigma_\mathrm{H}$

3.1. Syntax

The syntax of $\mathrm{F}^\sigma_\mathrm{H}$ 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. For each B, we fix a set $\mathcal{K}_B$ of the constants in that type; we require the typing rules for constants and the typing and evaluation rules for operations to respect this set. We also require that $\mathcal{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$. 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 $[v/x]e$ reduces to true. As mentioned before, refinement types in $\mathrm{F}^\sigma_\mathrm{H}$ is notable relative to existing manifest calculi in that *any* type (even a refinement type) can be refined, not just base types (as in [Flanagan 2006; Greenberg et al. 2010; Gronski and Flanagan 2007; Knowles and Flanagan 2010; Ou et al. 2004]).

In the syntax of terms, the first line is standard for a call-by-value polymorphic language: variables, constants, several monomorphic first-order operations op (i.e., destructors of one or more base-type arguments), term and type abstractions, and term and type applications. Note that there is no value restriction on type abstractions—as in System F, we do not evaluate under type abstractions, so there is no issue with or-

[2]Precisely speaking, in [Flanagan 2006; Greenberg et al. 2010], all base types come with predicate contracts. So, a "base type" here means that its contract is always true.

dering of effects. The second line offers the standard constructs of a manifest contract calculus [Flanagan 2006; Greenberg et al. 2010; Knowles and Flanagan 2010], with a few alterations, discussed below.

Casts are the distinguishing feature of manifest contract calculi. Unlike other manifest calculi, casts in $\mathrm{F}_{\mathrm{H}}^{\sigma}$ take substitutions (ranged over by σ), which are finite mappings from term and type variables to terms and types, respectively. The cast $\langle T_1 \Rightarrow T_2\rangle^{l}_{\sigma}$ ensures that, when applied to a value of type $\sigma(T_1)$, its argument behaves—and is treated—like a value of type $\sigma(T_2)$. The substitution σ is called a *delayed substitution* because substitution (triggered by, say, β-reduction) does not propagate into casts, at which it is recorded as delayed. A delayed substitution is applied when refinements in a cast are checked at runtime. $\mathrm{F}_{\mathrm{H}}^{\sigma}$ uses delayed substitutions to *statically* determine how casts work. In fact, as we will see below in Section 3.2, the reduction rules for a cast are selected independently of any terms or types substituted in the cast's source and target types, unlike F_{H}. This is crucial to prove the property that we call *cotermination* (Lemma 5.5), a false conjecture in prior work.

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 labels here are drawn from an arbitrary set, in practice l will refer to a source-code location.) Finally, we use active checks $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^{l}$ to support a small-step semantics for checking casts into refinement types. In an active check, $\{x{:}T \mid e_1\}$ is the refinement being checked, e_2 is the current state of checking, and v is the value being checked. The type in the first position of an active check is not necessary for the operational semantics, but we keep it around as a technical aid to type soundness. The value in the third position can be any value, not just a constant according to generalization of refinement types. 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 present scope.)

The values in $\mathrm{F}_{\mathrm{H}}^{\sigma}$ are constants, term and type abstractions, and casts. We also define *results*, which are either values or blame. Type soundness, stated in Theorem 5.19, will show that evaluation produces a result, but not necessarily a value. We note that, unlike some contract calculi—i.e., blame calculus [Wadler and Findler 2009]—, function cast applications $\langle x{:}T_{11} \rightarrow T_{12} \Rightarrow x{:}T_{21} \rightarrow T_{22}\rangle^{l}\, v$ are not seen as values, which simplifies our inversion lemmas. Instead, casts between function types will η-expand and wrap with the casts on the domain and the codomain their argument. This makes the notion of "function proxy" explicit: the cast semantics adds many new closures.

To define semantics, we use evaluation contexts [Felleisen and Hieb 1992] (ranged over by E), a standard tool to introduce small-step operational semantics. The syntax of evaluation contexts shown in Figure 1 means that the semantics evaluates subterms from left to right in the call-by-value style.

As usual, we introduce some conventional notations. We write $\mathrm{FV}(e)$ (resp. $\mathrm{FV}(T)$) to denote free term variables in the term e (resp. the type T), which is defined as usual, except for casts:

$$\mathrm{FV}(\langle T_1 \Rightarrow T_2\rangle^{l}_{\sigma}) = ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma)$$

where $\mathsf{dom}(\sigma)$ is the domain set of σ and $\mathrm{FV}(\sigma)$ is the set of free term variables in terms and types mapped by σ. Similarly, we use $\mathrm{FTV}(e)$, $\mathrm{FTV}(T)$ and $\mathrm{FTV}(\sigma)$ for free type variables, and $\mathrm{AFV}(e)$, $\mathrm{AFV}(T)$ and $\mathrm{AFV}(\sigma)$ for all free variables, namely, both free term and type variables. We say that terms and types are closed when they have no free term and type variables.

We define application of substitutions, which is almost standard except the case for casts, below. To preserve standard properties of substitution, such as, "applying a substitution to a closed term yields the same term," we consider only terms without garbage bindings in delayed substitutions and assume that $\mathrm{dom}(\sigma) \subseteq \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)$ holds for every cast $\langle T_1 \Rightarrow T_2 \rangle^l_\sigma$. Before defining application of substitution, we introduce a few auxiliary notations. For a set S of variables, $\sigma \mid S$ denotes the restriction of σ to S. Formally,

$$\sigma \mid S = (\{x \mapsto \sigma(x) \mid x \in \mathrm{dom}(\sigma) \cap S\}, \{\alpha \mapsto \sigma(\alpha) \mid \alpha \in \mathrm{dom}(\sigma) \cap S\}).$$

We denote by $\sigma_1 \uplus \sigma_2$ a delayed substitution obtained by concatenating substitutions elementwise.

3.1 Definition [Substitution]: Substitution in $\mathrm{F}^\sigma_\mathrm{H}$ is the standard substitution function with a single change, in the cast case:

$$\sigma(\langle T_1 \Rightarrow T_2 \rangle^l_{\sigma_1}) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma_2}$$

where $\sigma_2 = \sigma(\sigma_1) \uplus (\sigma \mid (\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1))$.

Here, $\sigma(\sigma_1)$ means application of σ to terms and types mapped by σ_1. Notice that the restriction on σ is required for the assumption on terms mentioned above. We will see that many properties of substitution in lambda calculi hold for our substitution later.

Finally, we introduce several syntactic shorthands. We write $T_1 \to T_2$ for $x{:}T_1 \to T_2$ when x does not appear free in T_2 and $\langle T_1 \Rightarrow T_2 \rangle^l$ for $\langle T_1 \Rightarrow T_2 \rangle^l_\sigma$ if the domain of σ is empty. A let expression $\mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2$ denotes an application term of the form $(\lambda x{:}T.\ e_2)\, e_1$. We may omit the type if it is clear from the context. If $\sigma = (\{x \mapsto e\}, \emptyset)$, then we write $[e/x]e'$, $[e/x]T'$ and $[e/x]\sigma'$ for $\sigma(e')$, $\sigma(T')$ and $\sigma(\sigma')$, respectively. Similarly, we write $[T/\alpha]e'$, $[T/\alpha]T'$ and $[T/\alpha]\sigma'$ for $\sigma(e')$, $\sigma(T')$ and $\sigma(\sigma')$, respectively, if $\sigma = (\emptyset, \{\alpha \mapsto T\})$.

3.2. 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 ($\longrightarrow$). We define these relations as over *closed* terms.

The latter relation is standard. The E_REDUCE rule lifts $\rightsquigarrow$ reductions into $\longrightarrow$; the E_COMPAT rule turns $\longrightarrow$ into a congruence over evaluation contexts; and the E_BLAME rule lifts blame, treating it as an uncatchable exception. The reduction relation $\rightsquigarrow$ is more interesting. There are four different kinds of reductions: the standard lambda calculus reductions, structural cast reductions, cast staging reductions, and checking reductions.

The E_BETA, and E_TBETA rules should need no explanation—these are the standard call-by-value polymorphic lambda calculus reductions. The E_OP rule uses a denotation function $[\![-]\!]$ to give meaning to the first-order operations. In Section 3.3, we describe a property of $[\![-]\!]$ to be required for showing type soundness.

The E_REFL, E_FUN, and E_FORALL rules reduce casts structurally. 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 6.) 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 [Greenberg et al. 2010] for an explanation). As one may notice, similarly to substitution (Definition 3.1), E_FUN and other cast rules restrict the domain of each delayed substitution in the right-hand side of reduction to free variables in the source and the target types of the corresponding cast.

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

$$
\begin{array}{rcll}
\mathrm{op}\,(v_1, \ldots, v_n) & \rightsquigarrow & [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n) & \text{E\_OP} \\
(\lambda x\colon T_1.\ e_{12})\, v_2 & \rightsquigarrow & [v_2/x]e_{12} & \text{E\_BETA} \\
(\Lambda\alpha.\ e)\ T & \rightsquigarrow & [T/\alpha]e & \text{E\_TBETA}
\end{array}
$$

$$
\begin{array}{rcll}
\langle T \Rightarrow T\rangle^l_\sigma\ v & \rightsquigarrow & v & \text{E\_REFL} \\
\langle x\colon T_{11} \to T_{12} \Rightarrow x\colon T_{21} \to T_{22}\rangle^l_\sigma\ v & \rightsquigarrow & & \text{E\_FUN}
\end{array}
$$
$$\lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ y : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\ x\ \mathsf{in}\ \langle [y/x]T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\,(v\ y)$$
$$\text{when } x\colon T_{11} \to T_{12} \neq x\colon T_{21} \to T_{22} \text{ and } x \notin \mathrm{dom}(\sigma) \text{ and}$$
$$y \text{ is fresh and, for } i \in \{1,2\},\ \sigma_i = \sigma \mid \mathrm{AFV}(T_{1i}) \cup \mathrm{AFV}(T_{2i})$$
$$\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l_\sigma\ v \rightsquigarrow \Lambda\alpha.\ (\langle [\alpha/\alpha]T_1 \Rightarrow T_2\rangle^l_\sigma\,(v\ \alpha)) \qquad \text{E\_FORALL}$$
$$\text{when } \forall\alpha.T_1 \neq \forall\alpha.T_2 \text{ and } \alpha \notin \mathrm{dom}(\sigma)$$

$$\langle\{x\colon T_1 \mid e\} \Rightarrow T_2\rangle^l_\sigma\ v \rightsquigarrow \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}\ v \qquad \text{E\_FORGET}$$
$$\text{when } T_2 \neq \{x\colon T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x\colon T_1 \mid e\} \mid e_2\}$$
$$(\sigma' = \sigma \mid \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2))$$
$$\langle T_1 \Rightarrow \{x\colon T_2 \mid e\}\rangle^l_\sigma\ v \rightsquigarrow \qquad \text{E\_PRECHECK}$$
$$\langle T_2 \Rightarrow \{x\colon T_2 \mid e\}\rangle^l_{\sigma_1}\,(\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\ v)$$
$$\text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x\colon T' \mid e'\}$$
$$(\sigma_1 = \sigma \mid \mathrm{AFV}(\{x\colon T_2 \mid e_2\}) \text{ and } \sigma_2 = \sigma \mid \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2))$$
$$\langle T \Rightarrow \{x\colon T \mid e\}\rangle^l_\sigma\ v \rightsquigarrow \langle\sigma(\{x\colon T \mid e\}), \sigma([v/x]e), v\rangle^l \qquad \text{E\_CHECK}$$

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

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

$$\frac{e_1 \rightsquigarrow e_2}{e_1 \longrightarrow e_2}\ \text{E\_REDUCE} \qquad \frac{e_1 \longrightarrow e_2}{E\,[e_1] \longrightarrow E\,[e_2]}\ \text{E\_COMPAT} \qquad \frac{}{E\,[\Uparrow l] \longrightarrow \Uparrow l}\ \text{E\_BLAME}$$

Fig. 2. Operational semantics for $\mathrm{F}_{\mathrm{H}}^{\sigma}$

Note that E_FUN uses a let expression—syntactic sugar for immediate application of a lambda—for the domain check. This is a nicer evaluation semantics than one in the previous calculi where the domain check can be duplicated by substitution. This way, some of our proofs are simplified. The E_FORALL rule is similar to E_FUN, generating a type abstraction with the necessary covariant cast. A seemingly trivial substitution $[\alpha/\alpha]$ is necessary for type soundness. The value v in this rule is expected to have $\forall\alpha.T_1$ and then $v\ \alpha$ is given type $[\alpha/\alpha]T_1$, which is not the same as T_1 in general since substitution is delayed at casts! So, after the reduction, the source type of the cast has to be $[\alpha/\alpha]T_1$. Side conditions on E_FORALL and E_FUN ensure that these rules apply only when E_REFL doesn't.

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

Before explaining how these rules interact in general, we offer a few examples. First, here is a reduction using E_CHECK, E_COMPAT, E_OP, and E_OK:

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

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

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

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

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

There are two cases where we need to use E_PRECHECK. First, when nested 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 a function or universal type is cast 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 6.)

We offer two higher-level ways to understand the interactions of these complicated cast rules. First, we can see the reduction rules as an unfolding of a recursive function, choosing the first clause in case of ambiguity. That is, the operational semantics unfolds a cast $\langle T_1 \Rightarrow T_2\rangle^l_\sigma\ v$ like $\mathcal{C}^l_\sigma(T_1, T_2, v)$:

$$\begin{aligned}
\mathcal{C}^l_\sigma(T, T, v) &= v \\
\mathcal{C}^l_\sigma(\{x{:}T_1 \mid e\}, T_2, v) &= \mathcal{C}^l_\sigma(T_1, T_2, v) \\
\mathcal{C}^l_\sigma(T_1, \{x{:}T_2 \mid e\}, v) &= \mathsf{let}\ x = \mathcal{C}^l_\sigma(T_1, T_2, v)\ \mathsf{in}\ \langle \sigma(\{x{:}T_2 \mid e\}), \sigma(e), x\rangle^l \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{where } x \notin \mathrm{dom}(\sigma)) \\
\mathcal{C}^l_\sigma(\forall\alpha.T_1, \forall\alpha.T_2, v) &= \Lambda\alpha.\ \mathcal{C}^l_\sigma(T_1, T_2, v\ \alpha) \qquad (\text{where } \alpha \notin \mathrm{dom}(\sigma)) \\
\mathcal{C}^l_\sigma(x{:}T_{11} \to T_{12}, x{:}T_{21} \to T_{22}, v) &= \\
\lambda x{:}T_{21}.\ \mathsf{let}\ y = \mathcal{C}^l_\sigma(T_{21}, T_{11}, x)\ &\mathsf{in}\ \mathcal{C}^l_\sigma([y/x]T_{12}, T_{22}, v\ y) \quad (\text{where } x \notin \mathrm{dom}(\sigma))
\end{aligned}$$

Alternatively, the rules firing during the evaluation of a cast in the small-step semantics obeys the following 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 $\langle T_1 \Rightarrow T_2\rangle^l\ v$ where we omit delayed substitution for conciseness. To simplify the following discussion, we define $\mathrm{unref}(T)$ as T without any outer refinements (though refinements on, e.g., the domain of a function would be unaffected); we write $\mathrm{unref}_n(T)$ when we remove only the n outermost refinements:

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

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

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

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

The E_REFL rule merits some more discussion. At first, it appears that we can dispense with this rule because a cast $\langle T \Rightarrow T\rangle^l_\sigma$ seems like it can't do anything: any value it applies must have already had type $\sigma(T)$, so what could go wrong during any checks? One might worry that adding such a cast will cause a different label to be blamed. What we would have to prove is contextual equivalence of $\langle T \Rightarrow T\rangle^l_\sigma$ and an identity function (in the absence of E_REFL), for example, by following Belo et al. [2011][3]. We haven't been able to prove parametricity (and the upcast lemma) for a system without E_REFL because our logical relation is defined for *untyped* terms.

3.3. Static typing

The type system comprises three mutually recursive judgments: context well formedness ($\vdash \Gamma$), type well formedness ($\Gamma \vdash T$), and term well typing ($\Gamma \vdash e : T$). The rules for contexts and types are unsurprising. The rules for terms are mostly standard.

[3]Note that the upcast lemma in Belo et al. [2011] is for a system with E_REFL and equivalence of $\langle T \Rightarrow T\rangle^l_\sigma$ and an identity function is trivial.

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash \sigma(T_1) \quad \Gamma \vdash \sigma(T_2) \quad T_1 \parallel T_2 \quad \mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(T_1) \to \sigma(T_2)}\ \textsc{T\_Cast}$$

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

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

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

Fig. 3. Typing rules for $\mathrm{F}_{\mathrm{H}}^{\sigma}$

First, the T_CONST and T_OP rules use the ty function to assign (possibly dependent) monomorphic first-order types to constants and operations, respectively; we require constants to satisfy the predicate (if any) of $\mathsf{ty}(k)$ and $[\![\mathrm{op}]\!]$ to be a function that returns a value satisfying the predicate of the codomain type of $\mathsf{ty}(\mathrm{op})$ when each argument value satisfies the predicate of the corresponding domain type of $\mathsf{ty}(\mathrm{op})$. The T_APP rule is dependent, to account for dependent function types. The T_CAST rule allows casts between compatibly structured well formed types, demanding that both source and target types after applying delayed substitution be well-formed. Compatibility of type structures is defined in Figure 4; intuitively, compatible types are identical when predicates in them are ignored. In particular, it is critical that type variables are compatible with only (refinements of) themselves because we have no idea what type will be substituted for α. If we allow type variable α to be compatible with another type, say, B, then the check with the cast from α to B would not work when α is replaced with a function type or a quantified type. Moreover, this definition helps us avoid nontermination due to non-parametric operations (e.g., Girard's J operator); it's imperative that a term like

$$\mathsf{let}\ \delta = \Lambda\alpha.\ \lambda x{:}\alpha.\ \langle \alpha \Rightarrow \forall\beta.\beta \to \beta\rangle^{l}\, \alpha\, x\ \mathsf{in}\ \delta\ \forall\beta.\beta \to \beta\ \delta$$

isn't well typed. Note that, in T_CAST, we assign casts a non-dependent function type and that we do not require well typedness/formedness of terms/types mapped by delayed substitution in a direct way—though well typed programs will start with and preserve well typed substitutions. Finally, it's critical that compatibility is substitutive, i.e., that if $T_1 \parallel T_2$, then $([e/x]T_1) \parallel T_2$ (Lemma A.27).

Some of the typing rules—T_CHECK, T_BLAME, T_EXACT, T_FORGET, and T_CONV—are "runtime only". These rules aren't needed to typecheck source programs, but we need them to guarantee preservation. T_CHECK, T_EXACT, and T_CONV are excluded from source programs because we don't want the typing of source programs to rely on the evaluation relation; such an interaction is acceptable in this setting, but disrupts the phase distinction and is ultimately incompatible with nontermination and effects. We exclude T_BLAME because programs shouldn't *start* with failures. Finally, we exclude T_FORGET because we imagine that source programs have all type changes explicitly managed by casts. 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 typecheck 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 $[v/x]e_1 \longrightarrow^{*} e_2$, which is necessary to avoid nonsensical terms like $\langle\{x{:}T \mid x \geq 0\}, \mathsf{true}, -1\rangle^{l}$, where the wrong predicate gets checked. The T_EXACT rule allows us to retype a closed value of type T at $\{x{:}T \mid e\}$ if $[v/x]e \longrightarrow^{*} \mathsf{true}$. 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 $[v/x]e_1 \longrightarrow^{*} \mathsf{true}$, so T_EXACT applies. Earlier systems used most-specific types and subtyping to show that the E_OK rule preserves typing. While the "most specific" requirement is abstract, a constant $k \in \mathcal{K}_B$ is typically given the *selfified* type $\mathsf{ty}(k) = \{x{:}B \mid x = k\}$ [Ou et al. 2004]. But functions don't admit a decidable equality, so there isn't an obvious way to assign them most-specific types. T_EXACT is a suitably extensional,syntactic, and subtyping-free replacement for the earlier semantic requirement: constants and functions can be assigned less specific types, but we can use T_EXACT in the preservation proof to remember successful checks.

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

$$\frac{}{\alpha \parallel \alpha}\ \textsc{Sim\_Var} \qquad \frac{}{B \parallel B}\ \textsc{Sim\_Base}$$

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

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

Conversion $\boxed{\sigma_1 \longrightarrow^* \sigma_2}$ $\boxed{T_1 \equiv T_2}$

$$\sigma_1 \longrightarrow^* \sigma_2 \iff \begin{array}{l}\mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2) \subset \mathsf{TmVar} \wedge \\ \forall x \in \mathsf{dom}(\sigma_1).\ \sigma_1(x) \longrightarrow^* \sigma_2(x)\end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ \textsc{C\_Var} \qquad \frac{}{B \equiv B}\ \textsc{C\_Base} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \equiv T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \equiv \{x{:}T_2 \mid \sigma_2(e)\}}\ \textsc{C\_Refine}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'}\ \textsc{C\_Fun} \qquad \frac{T \equiv T'}{\forall\alpha.T \equiv \forall\alpha.T'}\ \textsc{C\_Forall}$$

$$\frac{T_2 \equiv T_1}{T_1 \equiv T_2}\ \textsc{C\_Sym} \qquad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}\ \textsc{C\_Trans}$$

Fig. 4. Type compatibility and conversion for $\mathrm{F}_{\mathrm{H}}^{\sigma}$

Finally, the T_CONV rule is introduced as a technical device to prove preservation. This rule is motivated by reduction of an function application: $v_1\ e_2 \longrightarrow v_1\ e_2'$. Here, the first term is typed at $[e_2/x]\,T_2$ (by T_APP), but reapplying T_APP types the second term at $[e_2'/x]\,T_2$. The T_CONV rule allows terms to be retyped at convertible types in order to type $v_1\ e_2'$ at $[e_2/x]\,T_2$. We define a conversion relation $\equiv$, which we also call *common-subexpression reduction*, or CSR, using rules in Figure 4. Roughly speaking, T_1 and T_2 are convertible when there is a common type T and subexpressions e_1 and e_2 of T_1 and T_2 such that $T_1 = [e_1/x]\,T$ and $T_2 = [e_2/x]\,T$ and $e_1 \longrightarrow^* e_2$. In the case of the function application above, T_2 is a common type and e_2 and e_2' are subexpressions to reduce. The only interesting rule is C_REFINE, which says that refinement types $\{x{:}T_1 \mid e_1\}$ and $\{x{:}T_2 \mid e_2\}$ are convertible when T_1 and T_2 are convertible and there are some substitutions σ_1, σ_2 and a common subexpression e such that $e_1 = \sigma_1(e)$ and $e_2 = \sigma_2(e)$ and each term mapped by σ_1 reduces to one mapped by σ_2 (this is the reason why we call the relation $\equiv$ CSR). We remark that this conversion relation is different from that given in the original ESOP 2011 work [Belo et al. 2011], where their conversion relation is defined in terms of parallel reduction. Unfortunately, however, it turns out that parallel reduction does not quite have the properties we need. We discuss this further in Section 7. Another remark is that Belo et al. [2011] also (falsely) claimed that symmetry of convertible relation was not necessary for type soundness or parametricity, but symmetry is in fact used in the proof of preservation (Lemma A.39, when a term typed by T_APP steps by E_REDUCE/E_REFL).

4. EXAMPLES

To better understand the semantics, we offer two examples: a closer look at the NAT datatype and a "library as a language" that uses contracts to implement a type system.

4.1. Contracts for abstract datatypes

The standard polymorphic encodings of existential and product types transfer over to $\mathrm{F}_{\mathrm{H}}^{\sigma}$'s System F-style impredicative polymorphism without a problem. Indeed, 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.

$$\begin{aligned}(x : T_1) \times T_2 &= \forall\alpha.(x{:}T_1 \to T_2 \to \alpha) \to \alpha \\ (e_1, e_2)_{(x:T_1)\times T_2} &= \Lambda\alpha.\ \lambda f{:}(x{:}T_1 \to T_2 \to \alpha).\ f\ e_1\ e_2\end{aligned}$$

As in pure System F, we need to specify types on the encoding of pairs—we'll omit these types when they are clear from context. We also use record projection notation and field names, to make the example clearer.

Let's return to our simple example combining contracts and polymorphism—an abstract datatype of natural numbers.

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

We omit the implementation, a standard Church encoding, where $\alpha = \forall\beta.\beta \to (\beta \to \beta) \to \beta$. The constructors zero and succ are standard; the operator iszero determines whether a natural is zero; the operator pred yields the predecessor. We can encode dependent sums as:

$$\begin{aligned}(x : T_1) \times T_2 &= \forall\alpha.(x{:}T_1 \to T_2 \to \alpha) \to \alpha \\ (e_1, e_2) &= \Lambda\alpha.\ \lambda f{:}(x{:}T_1 \to T_2 \to \alpha).\ f\ e_1\ e_2\end{aligned}$$

As we saw above, the standard representation the naturals is *inadequate* with respect to the mathematical natural numbers, in particular with respect to pred . In math, $\mathsf{pred}\ \mathsf{zero}$ is undefined, but the implementation will return zero. The NAT interface hides our encoding of the naturals behind an existential type, but to ensure adequacy, we want to ensure that pred is only ever applied to terms of type $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\, x)\}$. With contracts, this is easy enough:

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

Recall that in the Church encoding, α will be instantiated with $\forall\beta.\beta \to (\beta \to \beta) \to \beta$. So the refinement $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\, x)\}$ in the new type of pred is a refinement of a polymorphic function type. These general refinement types are available in $\mathrm{F}_{\mathrm{H}}^{\sigma}$, but they were not in earlier manifest calculi.

To see why this more specific type for pred is useful, consider the following expression.

$$\mathsf{unpack}\ \mathsf{NAT} : \exists\alpha.\ I\ \mathsf{as}\ \alpha, n\ \mathsf{in}\ n.\mathsf{iszero}\,(n.\mathsf{pred}\,(n.\mathsf{zero})) : \mathsf{Bool}$$

Here I is the interface we specified for NAT. We've "unpacked" the ADT to make its type available as α; its constructors and operators are in the dependent pair n. We then ask if the predecessor of 0 is 0, running $n.\mathsf{iszero}\,(n.\mathsf{pred}\,(n.\mathsf{zero}))$. The inner application is *not well typed*! We have that $\mathsf{zero} : \alpha$, but the domain type of pred is $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\, x)\}$. In order to make the application well typed, we must insert a cast:

$$\begin{aligned}&\mathsf{unpack}\ \mathsf{NAT} : \exists\alpha.\ I\ \mathsf{as}\ \alpha, n\ \mathsf{in} \\ &\quad n.\mathsf{iszero}\,(n.\mathsf{pred}\,(\langle \alpha \Rightarrow \{x{:}\alpha \mid \mathsf{not}\,(n.\mathsf{iszero}\, x)\}\rangle^{l}\ n.\mathsf{zero})) : \mathsf{Bool}\end{aligned}$$

Naturally, this cast will ultimately raise $\Uparrow l$, because $\mathsf{not}\,(n.\mathsf{iszero}\; n.\mathsf{zero}) \longrightarrow^* \mathsf{false}$.

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 datatype, consider the extension of the interface with a subtraction operation, sub, and a binary "less than or equal" operator, leq. Natural number subtraction $\mathsf{sub}\; x\; y$ is defined only when $\mathsf{leq}\; y\; x$; we can specify this pre-condition as before, by refining the type of sub's second argument. But subtraction comes with a guarantee, as well: $\mathsf{sub}\; x\; y$ will always be less than or equal to x. That is, the result $\mathsf{sub}\; x\; y$ has the refined type $\{z{:}\alpha \mid \mathsf{leq}\; z\; x\}$. We can specify both of these facts with the interface:

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

The sub function's contract requires that sub's second argument is less than or equal to the first; the contract requires that sub returns a result that is less than or equal to the first argument.

How can we write an implementation to meet this interface? By putting casts in the implementations. We can impose the contracts on pred and sub when we "pack up" the implementation NAT. Writing nat for the type of the Church encoding $\forall\beta.\beta \to (\beta \to \beta) \to \beta$, we define the exported pred and sub in terms of the standard, unrefined implementations, pred' and sub' .

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

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, because we're casting out of a refinement. 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 abstract datatype interfaces as follows: the positive parts of the interface type are checked by the datatype's contract and can raise blame—these parts are the responsibility of the ADT's implementation; the negative parts of the interface type are not checked by the datatype's contract—these parts are the responsibility of the ADT's clients. Distributing obligations in this way recalls Findler and Felleisen's seminal idea of client and server blame [Findler and Felleisen 2002].

4.2. Contracts as type systems

Inasmuch as abstract datatypes are little languages, contracts for abstract datatypes are like type systems for little languages, following the ideas in Harper et al. [1993]. In this section, we develop a toy combinator language of string transducers, which specify mappings between regular languages.[4] A type system for the transducer combinators translates naturally to a contracted abstract datatype implementation.

Each transducer $t : \mathcal{L}_1 \hookrightarrow \mathcal{L}_2$ maps from a (regular) domain $\mathsf{dom}\; t = \mathcal{L}_1$ to a (regular) range $\mathsf{rng}\; t = \mathcal{L}_2$.

$$\begin{aligned} \mathcal{S} &::= \text{strings} \\ \mathcal{L} &::= \text{regular expressions} \\ t &::= \mathsf{copy}\,\mathcal{L} \mid \mathsf{delete}\,\mathcal{L} \mid \mathsf{concat}\; t_1\; t_2 \mid \mathsf{seq}\; t_1\; t_2 \end{aligned}$$

[4]This is effectively a unidirectional version of Boomerang [Bohannon et al. 2008].

Let ϵ be the empty string, and let $\cdot$ be string concatenation. The combinators compose to specify transducers. For this example, we only need two primitive combinators: $\mathsf{copy}\,\mathcal{L}$, maps a string in the language $\mathcal{L}$ to itself; and $\mathsf{delete}\,\mathcal{L}$ which maps a string in the language $\mathcal{L}$ to the empty string, ϵ. We can combine transducers in two ways: $\mathsf{concat}\ t_1\ t_2$ runs t_1 on the first part of the input and t_2 on the second; $\mathsf{seq}\ t_1\ t_2$ runs t_2 on the output t_1. We can define a semantics for transducers easily enough, with a function $\mathsf{run}\ t$ that takes strings in $\mathsf{dom}\ t$ to strings in $\mathsf{rng}\ t$.

The $\mathsf{copy}\,\mathcal{L}$ transducer has the simplest semantics: it just copies strings in the given language, $\mathcal{L}$. Its typing rule is straightforward.

$$\mathsf{run}\,(\mathsf{copy}\,\mathcal{L})\,\mathcal{S} = \mathcal{S} \qquad\qquad \mathsf{copy}\,\mathcal{L} : \mathcal{L} \hookrightarrow \mathcal{L}$$

The $\mathsf{delete}\,\mathcal{L}$ transducer deletes a string in the language $\mathcal{L}$; its typing rule indicates that its range is the language containing only the empty string, $\{\epsilon\}$.

$$\mathsf{run}\,(\mathsf{delete}\,\mathcal{L})\,\mathcal{S} = \epsilon \qquad\qquad \mathsf{delete}\,\mathcal{L} : \mathcal{L} \hookrightarrow \{\epsilon\}$$

The $\mathsf{concat}\ t_1\ t_2$ combinator splits its input between its two sub-transducers. Splitting up the input makes the semantics somewhat subtle: in general, $\mathcal{S} \in \mathsf{dom}\ t_1 \cdot \mathsf{dom}\ t_2$ does not imply that there is a *unique* way to split $\mathcal{S}$. When two regular languages always split uniquely, we say they are *unambiguously splittable*, written $\mathcal{L}_1 \cdot^! \mathcal{L}_2$. Unambiguous splittability of regular languages is decidable [Bohannon et al. 2008]; if we only concatenate transducers with unambiguously splittable domains, then the run function will be unambiguous.

$$\begin{array}{l} \mathsf{run}\,(\mathsf{concat}\ t_1\ t_2)\ (\mathcal{S}_1 \cdot \mathcal{S}_2) = \\ \quad \mathsf{run}\ t_1\ \mathcal{S}_1 \cdot \mathsf{run}\ t_2\ \mathcal{S}_2 \\ \quad \text{where } \mathcal{S}_i \in \mathsf{dom}\ t_i \end{array} \qquad \frac{\begin{array}{c} t_1 : \mathcal{L}_{11} \hookrightarrow \mathcal{L}_{12} \\ t_2 : \mathcal{L}_{21} \hookrightarrow \mathcal{L}_{22} \quad \mathcal{L}_{11} \cdot^! \mathcal{L}_{21} \end{array}}{\mathsf{concat}\ t_1\ t_2 : \mathcal{L}_{11} \cdot \mathcal{L}_{21} \hookrightarrow \mathcal{L}_{12} \cdot \mathcal{L}_{22}}$$

Finally, the $\mathsf{seq}\ t_1\ t_2$ combinator runs t_2 on the output of t_1. The typing rule requires that the two sub-transducers match: $\mathsf{rng}\ t_1$ and $\mathsf{dom}\ t_2$ must be the same language.

$$\mathsf{run}\,(\mathsf{seq}\ t_1\ t_2)\,\mathcal{S} = \mathsf{run}\ t_2\,(\mathsf{run}\ t_1\ \mathcal{S}) \qquad\qquad \frac{t_1 : \mathcal{L}_1 \hookrightarrow \mathcal{L}_2 \qquad t_2 : \mathcal{L}_2 \hookrightarrow \mathcal{L}_3}{\mathsf{seq}\ t_1\ t_2 : \mathcal{L}_1 \hookrightarrow \mathcal{L}_3}$$

In order to facilitate programming with these transducers, we may try to embed these combinators inside of a functional programming. It is rather difficult to generalize this combinator language—typing a lambda calculus with string regular expression types is not easy [Tabuchi et al. 2003; Benzaken et al. 2003; Bierman et al. 2010]. It is easy, however, to define an abstract datatype offering these operations, assuming we have a type String of strings and Regex of regular expressions, with appropriate decision procedures for unambiguous splittability and equality.

$$\begin{array}{rl} \mathsf{TRANS} : \exists\alpha. & (\mathsf{dom} : \alpha \to \mathsf{Regex}) \times (\mathsf{rng} : \alpha \to \mathsf{Regex}) \times \\ & (\mathsf{run} : (\alpha \to \mathsf{String} \to \mathsf{String})) \times \\ & (\mathsf{copy} : \mathsf{Regex} \to \alpha) \times (\mathsf{delete} : (\mathsf{Regex} \to \alpha)) \times \\ & (\mathsf{concat} : \alpha \to \alpha \to \alpha) \times (\mathsf{seq} : (\alpha \to \alpha \to \alpha)) \end{array}$$

There is a problem, though: this datatype will let us write nonsensical combinators. In particular, we can give concat transducers that don't have unambiguously splittable domains, or we can give seq transducers which don't match up. For example, suppose $\epsilon \notin \mathcal{L}$ and $\mathcal{S} \in \mathcal{L}$. Let $t = \mathsf{seq}\,(\mathsf{delete}\,\mathcal{L})\,(\mathsf{copy}\,\mathcal{L})$. Then:

$$\mathsf{run}\,(\mathsf{seq}\,(\mathsf{delete}\,\mathcal{L})\,(\mathsf{copy}\,\mathcal{L}))\,\mathcal{S} = \epsilon$$

Since $\epsilon \notin \mathcal{L}$, this means that run took a value in $\mathsf{dom}\ t = \mathcal{L}$ and produced a value outside of $\mathsf{rng}\ t = \mathcal{L}$.

We are in a difficult position: we have a little language and a type system. But scaling our transducer language's type system up to the lambda calculus increases complexity, and distracts us from what we'd like to be doing—writing transducer programs!

Contracts offer a middle way. By putting contracts on the TRANS interface, we can ensure that all transducers are well formed.

$$\begin{aligned}\mathsf{TRANS} : \exists\alpha.\ & (\mathsf{dom} : \alpha \to \mathsf{Regex}) \times (\mathsf{rng} : \alpha \to \mathsf{Regex}) \times \\ & (\mathsf{run} : (t{:}\alpha \to \{x{:}\mathsf{String} \mid x \in \mathsf{dom}\ t\} \to \{x{:}\mathsf{String} \mid x \in \mathsf{rng}\ t\})) \times \\ & (\mathsf{copy} : \mathsf{Regex} \to \alpha) \times (\mathsf{delete} : (\mathsf{Regex} \to \alpha)) \times \\ & (\mathsf{concat} : (t_1{:}\alpha \to \{t_2{:}\alpha \mid \mathsf{splittable}\,(\mathsf{dom}\ t_1)\ (\mathsf{dom}\ t_2)\} \to \alpha)) \times \\ & (\mathsf{seq} : (t_1{:}\alpha \to \{t_2{:}\alpha \mid \mathsf{rng}\ t_1 = \mathsf{dom}\ t_2\} \to \alpha))\end{aligned}$$

The TRANS abstract datatype defines an embedded domain-specific language—with its own domain-specific type system. For example, concat will only accept transducers with unambiguously splittable domains, as discussed above; seq will only sequence transducers that match up. The interface given here is only one of many: it checks inputs but not outputs. For example, we could ensure that concat has our intended behavior by giving its codomain the type $\{t_3{:}\alpha \mid (\mathsf{dom}\ t_3 = (\mathsf{dom}\ t_1) \circ (\mathsf{dom}\ t_2)) \wedge (\mathsf{rng}\ t_3 = (\mathsf{rng}\ t_1) \circ (\mathsf{rng}\ t_2))\}$, where $\circ$ denotes regular expression concatenation.

The checks on run and seq are easy enough to build into our TRANS implementation. But there is a distinct advantage to making the contracts explicit in the interface type: the type system will keep track of unambiguous splittability checks. Programmers can track relevant information in refinements in client modules, and we can statically eliminate redundant checks via, e.g., subtyping.

In general, contracting abstract datatype interfaces allows for library designers to extend the language's type system with library-specific constraints. Clients then have two choices: propagate the library's contracts through their code, possibly avoiding redundant checks; or ignore the contracts within their own code, allowing the checks to happen whenever they call into the library. Either way, the library's users can rest assured that the contracts will guarantee the safety properties the library designers desired.

If programmers are careful to program in a "cover your ass" (CYA) style, wherein each library's interface uses contracts that are strong enough to guarantee that other libraries' contracts are satisfied, then error messages greatly improve. When libraries are stacked in a hierarchy several levels deep, CYA contracts in interfaces give programmers error messages earlier and at a higher level of abstraction.

As a final note before we begin the technical content: the foregoing is the current implementation strategy for Boomerang [Bohannon et al. 2008], a language of bidirectional string transducers called *lenses*. The semantic constraints on Boomerang combinators are decidable, but combining Boomerang's typing rules with the lambda calculus would be cumbersome—a hard open problem. Boomerang is a complex language, and there is no room in the "complexity budget" for a statically checked type system: lenses can already be difficult to program with and understand, and the complicated constraints necessary for type checking will only add more to the programmer's burden. Instead, the Boomerang primitives have contracts that ensure that they produce sane bidirectional transformations. The Boomerang libraries built atop these primitives have contracts as well, in a CYA style. Even without optimizations to reduce the number of dynamic checks, this improvement in error handling has proved quite useful. Contracts are particularly suited to the phased nature of Boomerang, since the contracts on Boomerang's lens combinators are "quasi-static". Lenses are constructed only once and then run many times. Running a lens merely requires checking regular language membership, so higher cost one-time checks can be amortized over many lens runs.

5. PROPERTIES OF F_H^σ

We show that well-typed programs don't get stuck—a well typed term evaluates to a result, i.e., a value or a blame (if evaluation terminates at all[5])—via preservation and progress [Wright and Felleisen 1994].

The following proof of type soundness is entirely syntactic, offering a new approach to manifest calculi. In typical formulations of manifest calculi [Greenberg et al. 2010; Flanagan 2006], subtyping is used instead of a conversion relation like our $\equiv$ relation; one of the contributions in this work is the insight that subtyping—with its accompanying metatheoretical complications that prevent a simple syntactic proof of type soundness—is not an essential component of manifest calculi.

As Greenberg [2013] and Sekiyama et al. [2015] have pointed out, the "value inversion" lemma (Lemma A.16), which says values typed at refinement types must satisfy their refinements, is a critical component of any sound manifest contract system, especially for proving progress. The type soundness proof in Belo et al. [2011] is missing this lemma—and can never have it, since its conversion relation does not enjoy cotermination. Greenberg [2013] leaves this lemma as a conjecture—which turns out to be false. This value inversion lemma is not merely a technical device to prove progress. Together with preservation and progress, it means that if a term typed at a refinement type evaluates to a value, then it satisfies the predicate of the type, giving a slightly stronger guarantee about well typed programs.

Perhaps surprisingly, the value inversion lemma is not trivial due to T_CONV: we must show that predicates of convertible refinement types are semantically equivalent. The proof of this property rests on cotermination (Lemma 5.5), which says that common-subexpression reduction does not change the behavior of terms. Cotermination also plays a significant role in showing parametricity. Finally, using these properties, we show progress (Theorem A.37) and preservation (Theorem A.39), which imply type soundness (Theorem 5.19).

5.1. Cotermination

Next, we show cotermination, which both type soundness and parametricity rest on. We discuss earlier conjectures concerning cotermination in Section 7; for the rest of this section, we offer the new, correct definitions without direct reference to old versions of F_H. We first show that cotermination holds in the most simple situation, namely, where the domain of substitutions is singleton, and then show cotermination. The key observation in proving cotermination is that terms substituted by CSR evaluate to ones substituted by the same CSR. We refer to implicitly determinism of the semantics.

5.1 Lemma [Determinism (Lemma A.5)]: If $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.

5.2 Lemma [Cotermination, left side (Lemma A.10)]: Suppose that $e_1 \longrightarrow e_2$. If $[e_1/x]e \longrightarrow e'$, then $[e_2/x]e \longrightarrow^* [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.

5.3 Lemma [Cotermination, right side (Lemma A.13)]: Suppose that $e_1 \longrightarrow e_2$. If $[e_2/x]e \longrightarrow e'$, then $[e_1/x]e \longrightarrow^* [e_1/x]e''$ for some e'' such that $e' = [e_2/x]e''$.

5.4 Lemma [Cotermination (Lemma A.14)]:— Suppose that $e_1 \longrightarrow e_2$.
(1) If $[e_1/x]e \longrightarrow^* \mathsf{true}$, then $[e_2/x]e \longrightarrow^* \mathsf{true}$.
(2) If $[e_2/x]e \longrightarrow^* \mathsf{true}$, then $[e_1/x]e \longrightarrow^* \mathsf{true}$.
— Suppose that $e_1 \longrightarrow^* e_2$.
(1) If $[e_1/x]e \longrightarrow^* \mathsf{true}$, then $[e_2/x]e \longrightarrow^* \mathsf{true}$.
(2) If $[e_2/x]e \longrightarrow^* \mathsf{true}$, then $[e_1/x]e \longrightarrow^* \mathsf{true}$.

[5]In fact, F_H^σ is terminating, as we will discover in Section 6.

5.5 Lemma [Cotermination at true]: Suppose that $\sigma_1 \longrightarrow^* \sigma_2$.

(1) If $\sigma_1(e) \longrightarrow^*$ true, then $\sigma_2(e) \longrightarrow^*$ true.
(2) If $\sigma_2(e) \longrightarrow^*$ true, then $\sigma_1(e) \longrightarrow^*$ true.

PROOF. By induction on the size of $\mathsf{dom}(\sigma_1)$ with Lemma A.14. □

5.2. Type soundness

Using cotermination, we show value inversion and then type soundness in a standard syntactic way.

5.6 Lemma [Cotermination of refinement types (Lemma A.15)]: If $\{x{:}T_1 \mid e_1\} \equiv \{x{:}T_2 \mid e_2\}$ then $T_1 \equiv T_2$ and $[v/x]e_1 \longrightarrow^*$ true iff $[v/x]e_2 \longrightarrow^*$ true, for all v.

5.7 Lemma [Value inversion (Lemma A.16)]: If $\emptyset \vdash v : T$ and $\mathrm{unref}_n(T) = \{x{:}T_n \mid e_n\}$ then $[v/x]e_n \longrightarrow^*$ true.

5.8 Lemma [Term substitutivity of conversion (Lemma A.22)]:
If $T_1 \equiv T_2$ and $e_1 \longrightarrow^* e_2$ then $[e_1/x]T_1 \equiv [e_2/x]T_2$.

5.9 Lemma [Type substitutivity of conversion (Lemma A.23)]:
If $T_1 \equiv T_2$ then $[T/\alpha]T_1 \equiv [T/\alpha]T_2$.

5.10 Lemma [Term weakening (Lemma A.24)]: 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 $\vdash \Gamma, x{:}T', \Gamma'$.

5.11 Lemma [Type weakening (Lemma A.25)]: 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 $\vdash \Gamma, \alpha, \Gamma'$.

5.12 Lemma [Term substitution (Lemma A.28)]: If $\Gamma \vdash e' : T'$, then

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

5.13 Lemma [Type substitution (Lemma A.31)]: If $\Gamma \vdash T'$ then

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

As is standard for type systems with conversion rules, we must prove inversion lemmas in order to reason about typing derivations in a syntax-directed way. We offer the statement of inversion for functions here; the rest are in Section A.2.

5.14 Lemma [Lambda inversion (Lemma A.32)]: 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) $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T)$.

Inversion lemmas in hand, we prove a canonical forms lemma to support a proof of progress. The canonical forms proof is "modulo" the unref function: the shape of the values of type $\{x{:}T \mid e\}$ are determined by the inner type T.

5.15 Lemma [Canonical forms (Lemma A.36)]: If $\emptyset \vdash v : T$, then:

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

5.16 Theorem [Progress (Theorem A.37)]: If $\emptyset \vdash e : T$, then either

(1) $e \longrightarrow e'$, or
(2) e is a result r, i.e., a value or blame.

The following regularity property formalizes an important property of the type system: all contexts and types involved are well formed. This is critical for the proof of preservation: when a term raises blame, we must show that the blame is well typed. With regularity, we can immediately know that the original type is well formed.

5.17 Lemma [Context and type well formedness (Lemma A.38)]: (1) If $\Gamma \vdash e : T$, then $\vdash \Gamma$ and $\Gamma \vdash T$.
(2) If $\Gamma \vdash T$ then $\vdash \Gamma$.

5.18 Theorem [Preservation (Theorem A.39)]: If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.

5.19 Theorem [Type Soundness]: If $\emptyset \vdash e : T$ and $e \longrightarrow^* e'$ and e' does not reduce, then e' is a result. Moreover, if $e' = v$ and $T = \{x{:}T'' \mid e''\}$, then $[v/x]e'' \longrightarrow^* \mathsf{true}$.

PROOF. The first half is shown by Theorems 5.16 and 5.18, and the second is by $\emptyset \vdash v : T$ and Lemma 5.7. $\square$

Requiring standard weakening, substitution, and inversion lemmas, the syntactic proof of type soundness is straightforward. Note that value inversion (Lemma 5.7) gives us a strong soundness property: if a term reduces to a value, it satisfies all of the predicates in its refinement types. It is easy to restrict $\mathrm{F}_\mathrm{H}^\sigma$ 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, which is needed to deal with nested refinement types. We don't give the proof here because it is subsumed by type soundness in $\mathrm{F}_\mathrm{H}^\sigma$.

6. PARAMETRICITY

We prove relational parametricity for three reasons: (1) it yields powerful reasoning techniques such as free theorems [Wadler 1989] and is the tool used for the upcast lemma [Belo et al. 2011]; (2) it indicates that contracts don't interfere with type abstraction, i.e., that F_H supports polymorphism in the same way that System F does; (3) we want to correct Belo et al. [2011] and Greenberg [2013]. The proof is mostly standard: we define a (syntactic) logical relation on terms and types, where each type is interpreted as a relation on terms and the relation at type variables is given as a parameter.

We begin by defining two relations: $r_1 \sim r_2 : T; \theta; \delta$ relates closed results, defined by induction on types; $e_1 \simeq e_2 : T; \theta; \delta$ relates closed expressions which evaluate to results in the first relation. The definitions are shown in Figure 5.[6] Both relations have three indices: a type T, a substitution θ for type variables, and a substitution δ for term

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

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

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

$$\begin{array}{rcl}
e_1 \simeq e_2 : T; \theta; \delta & \iff & \exists r_1 r_2, e_1 \longrightarrow^* r_1 \wedge e_2 \longrightarrow^* r_2 \wedge r_1 \sim r_2 : T; \theta; \delta
\end{array}$$

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

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

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

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

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

Fig. 5. The logical relation for parametricity

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. A term substitution δ maps from variables to pairs of closed values. We write projections δ_i $(i = 1, 2)$ to denote projections from this pair. We similarly write θ_i $(i = 1, 2)$ for a substitution that maps a type variable α to T_i where $\theta(\alpha) = (R, T_1, T_2)$.

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. We require that both values satisfy their refinements rather than having the first satisfy the predicate iff the second does because I want to know that values related at refinement types *actually inhabit those types*, i.e., actually satisfy the predicates of the refinement. The $\sim$ relation on results is extended to the relation $\simeq$ on closed terms in a straightforward manner: terms are related if and only if they both terminate at related results. Divergent terms are not related to each other—though we will discover that divergent terms do not exist in F_H. 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 (well-typed) casts yield related results when applied to related inputs, we also need a relation on types $T_1 \simeq T_2 : *; \theta; \delta$; we define this relation in Figure 5. We can use the logical relation on terms to handle the arguments of function types and refinement types. Note that the T_1 and T_2 in this relation are not necessarily closed; terms in refinement types, which should be related at Bool, are closed by applying substitutions. In the function and refinement type cases, the relation on a smaller type is universally quantified over logically related values. There are two choices of the type at which they should be related (for example, the second line of the function type case could change T_{11} to T_{21}). It does not really matter which side we choose, since they are related types. We are "left-leaning". Finally, we lift the type relation to open terms, writing $\Gamma \vdash T_1 \simeq T_2 : *$ when two types are equivalent for any Γ-respecting substitutions.

It is worth discussing two points peculiar to this formulation: terms in the logical relation are untyped, and the type indices are open.

We allow any relation on terms to be used in θ; terms related at T need not be well typed at T. The standard formulation of a logical relation is well typed throughout, requiring that the relation R in every triple be well typed, only relating values of type T_1 to values of type T_2 (e.g., [Pitts 2005a]). We suspect that part of the reason this proof of parametricity works is its similarity to Girard's untyped reducibility candidates. We have two motivations for leaving the relations untyped. First, functions of type $x{:}T_1 \rightarrow T_2$ must map related values ($v_1 \sim v_2 : T_1$) to related results...but at what type? While $[v_1/x]\,T_2$ and $[v_2/x]\,T_2$ are related in the type relation, terms that are well typed at one type won't necessarily be well typed at the other, whether definitions are left- or right-leaning. Second, this parametricity relation is design to that upcasts have no effect: if $T_1 <: T_2$, then $\langle T_1 \Rightarrow T_2 \rangle^l \sim \lambda x{:}T_1.\ x : T_1 \rightarrow T_2$. That is, we want a cast $\langle T_1 \Rightarrow T_2 \rangle^l$, of type $T_1 \rightarrow T_2$, to be related to the identity $\lambda x{:}T_1.\ x$, of type $T_1 \rightarrow T_1$. There is one small hitch: $\lambda x{:}T_1.\ x$ has type $T_1 \rightarrow T_1$, not $T_1 \rightarrow T_2$! 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.

The type indices of the term relation are not necessarily closed. Instead, 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 $x{:}T_1 \rightarrow T_2$: the *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 $v_1\, v_1' \simeq v_2\, v_2'$ at some type. The question here is which type index we should use. If we keep 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 [Knowles and Flanagan 2010] 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 this 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 the original proof [Belo et al. 2011], we found that the E_REFL rule $\langle T \Rightarrow T \rangle^l\, v \rightsquigarrow v$ is not just a convenient way to skip decomposing 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 this syntactic setting. On the one hand, the evaluation of well-typed programs never encounters casts with uninstan-

Complexity of casts

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

Fig. 6. Complexity of casts

tiated type variables—a key property of our evaluation relation. On the other hand, by parametricity, we expect every value of type $\forall \alpha . \alpha \to \alpha$ to behave the same as the polymorphic identity function. One of the values of this type is $\Lambda \alpha.\ \langle \alpha \Rightarrow \alpha \rangle^l$. Without E_REFL, however, applying this type abstraction to a compound type, say $\mathsf{Bool} \to \mathsf{Bool}$, and a function f of type $\mathsf{Bool} \to \mathsf{Bool}$ would return, by E_FUN, a wrapped version of f that is syntactically different from the f we passed in—that is, the function broke parametricity! We expect the returned value should behave the same as the input, though—the results are just *syntactically* different. 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.

Now we can set about proving parametricity (Lemma A.47). We begin with compositionality theorems relating the closing substitutions θ and δ to substitution in terms (Lemma A.40) and types (Lemma A.43); convertibility shows that our logical relation relates terms at convertible types (Lemma A.44); after some lemmas about casts and a separate induction relating casts between related types (Lemma A.46), we prove parametricity.

6.1 Lemma [Term compositionality (Lemma A.40)]: 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 $r_1 \sim r_2 : [e/x]\, T; \theta; \delta$.

6.2 Lemma [Type compositionality (Lemma A.43)]:
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ iff $r_1 \sim r_2 : [T'/\alpha]\, T; \theta; \delta$.

6.3 Lemma [Convertibility (Lemma A.44)]: If $T_1 \equiv T_2$ then $r_1 \sim r_2 : T_1; \theta; \delta$ iff $r_1 \sim r_2 : T_2; \theta; \delta$.

Before we can show parametricity (Lemma A.47), we prove in a separate induction that casts between related types are related (Lemma A.46).

We show that (well typed) casts relate to themselves by induction a cast complexity metric, cc, defined in Figure 6. The complexity of a cast is the number of steps it and its subparts can take. This definition is carefully dependent on our definition of type compatibility and our cast reduction rules. Doing induction on this metric greatly simplifies the proof: we show that stepping casts at related types yields either related non-casts, or lower complexity casts between related types. Note that we omit the σ, since the evaluation of casts *doesn't depend on delayed substitutions*. It may be easier for the reader to think of $cc(\langle T_1 \Rightarrow T_2 \rangle^l)$ as a three argument function—taking two types and a blame label—rather than a single argument function taking a cast.

6.4 Lemma [Cast reflexivity (Lemma A.46)]: If $\vdash \Gamma$ and $T_1 \parallel T_2$ and $\Gamma \vdash \sigma(T_1) \simeq \sigma(T_1) : *$ and $\Gamma \vdash \sigma(T_2) \simeq \sigma(T_2) : *$ and $\mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(\_{:}T_1 \to T_2)$.

Finally, we can prove relational parametricity—every well-typed term (under Γ) is related to itself for any Γ-respecting substitutions.

6.5 Theorem [Parametricity (Theorem A.47)]: (1) If $\Gamma \vdash e : T$ then $\Gamma \vdash e \simeq e : T$, and
(2) If $\Gamma \vdash T$ then $\Gamma \vdash T \simeq T : *$.

We refer readers to [Sekiyama and Igarashi 2014] for a significantly expanded account of parametricity for $\mathrm{F_H}$ *with recursion*. There, they have proved that their logical relation based on $\top\top$-closure [Pitts 2005b] is sound[7] with respect to contextual equivalence. Since the details of their technical developments are different from what we present here, we can only conjecture that our logical relation is also sound with respect to contextual equivalence.

We do have that logically related programs are by definition *behaviorally equivalent*: if $\emptyset \vdash e_1 \simeq e_2 : T$, then e_1 and e_2 coterminate at related results. When the results are constants or blame, the results are not only logically related, but equal.

6.1. Subtyping

We elide the proofs of subtyping—we believe those in Belo et al. [2011] and Greenberg et al. [2010] adapt straightforwardly, since the parametricity relation hasn't materially changed in $\mathrm{F}^\sigma_\mathrm{H}$. Sekiyama and Igarashi [2014] offer a complete account of an $\mathrm{F_H}$-like system with recursion, a parametricity relation that has a clear relationship to contextual equivalence, and proofs of subtyping.

7. THREE VERSIONS OF $\mathrm{F_H}$

We discuss comparison of $\mathrm{F}^\sigma_\mathrm{H}$ with two prior formulations of $\mathrm{F_H}$ without delayed substitution: Belo et al. [2011] from ESOP 2011 and Greenberg's thesis [Greenberg 2013]. Both of these defined variants of $\mathrm{F_H}$, showing type soundness, parametricity and upcast elimination. All of these results depend on two properties of the $\mathrm{F_H}$ type conversion relation: substitutivity (Lemma A.22) and cotermination (Lemma 5.5).

7.1. $\mathrm{F_H}$ 1.0: Belo et al. [2011]

Belo et al. [2011] got rid of subtyping and explicitly used the symmetric, transitive closure of parallel reduction $\Rrightarrow$ (Figure 7) as the conversion relation. (Parallel reduction is reflexive by definition.) The use of parallel reduction is inspired by Greenberg et al. [2010], in which type soundness of λ_H is proved by using cotermination and another property called *substitutivity* (if $e_1 \Rrightarrow e_2$ and $e_1' \Rrightarrow e_2'$ then $[e_1'/x]e_1 \Rrightarrow [e_2'/x]e_2$) of parallel reduction. These properties were needed also for type soundness of $\mathrm{F_H}$. Unfortunately, it turns out that parallel reduction in $\mathrm{F_H}$ is *not* substitutive—the proof was wrong—and cotermination, which was left as a conjecture ([Belo et al. 2011], p. 15), does not hold, either. Figure 8 offers three counterexamples: two to substitutivity, and one to both substitutivity and cotermination.

Why doesn't substitutivity hold in $\mathrm{F_H}$, when it did (so easily) in λ_H? Sources of the trouble are that (1) the $\mathrm{F_H}$ cast rules depend upon certain (syntactic) equalities between types and that (2) parallel reduction is defined over open terms. As a result, substitution may change reduction rules to be applied—both counterexamples to substitutivity in Figure 8 take advantage of it.

[7] And also complete, under certain conditions.

Parallel term reduction $\boxed{e_1 \Rrightarrow e_2}$

$$\frac{v_i \Rrightarrow v_i'}{\mathrm{op}\,(v_1,\,\ldots,\,v_n) \Rrightarrow [\![\mathrm{op}]\!]\,(v_1',\,\ldots,\,v_n')}\ \textsc{EP\_ROp} \qquad \frac{e_{12} \Rrightarrow e_{12}' \quad v_2 \Rrightarrow v_2'}{(\lambda x{:}T.\ e_{12})\ v_2 \Rrightarrow [v_2'/x]e_{12}'}\ \textsc{EP\_RBeta}$$

$$\frac{e \Rrightarrow e' \quad T_2 \Rrightarrow T_2'}{(\Lambda\alpha.\ e)\ T_2 \Rrightarrow [T_2'/\alpha]e'}\ \textsc{EP\_RTBeta} \qquad \frac{v \Rrightarrow v'}{\langle T \Rightarrow T\rangle^l\ v \Rrightarrow v'}\ \textsc{EP\_RRefl}$$

$$\frac{T_2 \neq \{x{:}T_1 \mid e\} \quad T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\} \quad T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2' \quad v \Rrightarrow v'}{\langle\{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\ v \Rrightarrow \langle T_1' \Rightarrow T_2'\rangle^l\ v'}\ \textsc{EP\_RForget}$$

$$\frac{T_1 \neq T_2 \quad T_1 \neq \{x{:}T \mid e\} \quad T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2' \quad e \Rrightarrow e' \quad v \Rrightarrow v'}{\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\ v \Rrightarrow \langle T_2' \Rightarrow \{x{:}T_2' \mid e'\}\rangle^l\ (\langle T_1' \Rightarrow T_2'\rangle^l\ v')}\ \textsc{EP\_RPreCheck}$$

$$\frac{T \Rrightarrow T' \qquad e \Rrightarrow e' \qquad v \Rrightarrow v'}{\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\ v \Rrightarrow \langle\{x{:}T' \mid e'\}, [v'/x]e', v'\rangle^l}\ \textsc{EP\_RCheck}$$

$$\frac{v \Rrightarrow v'}{\langle\{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \Rrightarrow v'}\ \textsc{EP\_ROK} \qquad \frac{}{\langle\{x{:}T \mid e_1\}, \mathsf{false}, v\rangle^l \Rrightarrow \Uparrow l}\ \textsc{EP\_RFail}$$

$$\frac{\begin{array}{l} x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22} \\ T_{11} \Rrightarrow T_{11}' \qquad T_{12} \Rrightarrow T_{12}' \qquad T_{21} \Rrightarrow T_{21}' \qquad T_{22} \Rrightarrow T_{22}' \qquad v \Rrightarrow v' \end{array}}{\begin{array}{l} \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\ v \Rrightarrow \\ \qquad \lambda x{:}T_{21}'.\ (\langle[\langle T_{21}' \Rightarrow T_{11}'\rangle^l\ x/x]\,T_{12}' \Rightarrow T_{22}'\rangle^l\ (v'\ (\langle T_{21}' \Rightarrow T_{11}'\rangle^l\ x))) \end{array}}\ \textsc{EP\_RFun}$$

$$\frac{\forall\alpha.T_1 \neq \forall\alpha.T_2 \qquad T_1 \Rrightarrow T_1' \qquad T_2 \Rrightarrow T_2' \qquad v \Rrightarrow v'}{\langle\forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\ v \Rrightarrow \Lambda\alpha.\ (\langle T_1' \Rightarrow T_2'\rangle^l\ (v'\ \alpha))}\ \textsc{EP\_RForall}$$

$$\frac{}{e \Rrightarrow e}\ \textsc{EP\_Refl} \qquad \frac{T_1 \Rrightarrow T_1' \quad e_{12} \Rrightarrow e_{12}'}{\lambda x{:}T_1.\ e_{12} \Rrightarrow \lambda x{:}T_1'.\ e_{12}'}\ \textsc{EP\_Abs} \qquad \frac{e_1 \Rrightarrow e_1' \quad e_2 \Rrightarrow e_2'}{e_1\ e_2 \Rrightarrow e_1'\ e_2'}\ \textsc{EP\_App}$$

$$\frac{e \Rrightarrow e'}{\Lambda\alpha.\ e \Rrightarrow \Lambda\alpha.\ e'}\ \textsc{EP\_TAbs} \qquad \frac{e_1 \Rrightarrow e_1' \quad T_2 \Rrightarrow T_2'}{e_1\ T_2 \Rrightarrow e_1'\ T_2'}\ \textsc{EP\_TApp}$$

$$\frac{e_i \Rrightarrow e_i'}{\mathrm{op}\,(e_1,\,\ldots,\,e_n) \Rrightarrow \mathrm{op}\,(e_1',\,\ldots,\,e_n')}\ \textsc{EP\_Op} \qquad \frac{T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2'}{\langle T_1 \Rightarrow T_2\rangle^l \Rrightarrow \langle T_1' \Rightarrow T_2'\rangle^l}\ \textsc{EP\_Cast}$$

$$\frac{T \Rrightarrow T' \quad e \Rrightarrow e'}{\langle T, e, k\rangle^l \Rrightarrow \langle T', e', k\rangle^l}\ \textsc{EP\_Check} \qquad \frac{}{E\,[\Uparrow l] \Rrightarrow \Uparrow l}\ \textsc{EP\_Blame}$$

Parallel type reduction $\boxed{T_1 \Rrightarrow T_2}$

$$\frac{}{T \Rrightarrow T}\ \textsc{EP\_TRefl} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \Rrightarrow T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \Rrightarrow \{x{:}T_2 \mid \sigma_2(e)\}}\ \textsc{EP\_TRefine}$$

$$\frac{T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2'}{x{:}T_1 \to T_2 \Rrightarrow x{:}T_1' \to T_2'}\ \textsc{EP\_TFun} \qquad \frac{T \Rrightarrow T'}{\forall\alpha.T \Rrightarrow \forall\alpha.T'}\ \textsc{EP\_TForall}$$

Fig. 7. Parallel reduction (for open terms).

Counterexample 1: substitutivity

Let T be a type with a free variable x.

$$\begin{aligned} e_1 &= \langle T \Rightarrow \{y{:}[5/x]\,T \mid \mathsf{true}\}\rangle^l\, 0 \\ e_2 &= \langle [5/x]\,T \Rightarrow \{y{:}[5/x]\,T \mid \mathsf{true}\}\rangle^l\, (\langle T \Rightarrow [5/x]\,T\rangle^l\, 0) \\ e_1' = e_2' &= 5 \end{aligned}$$

Observe that $e_1' \Rrightarrow e_2'$ (by EP_REFL) and $e_1 \Rrightarrow e_2$ (by EP_RPRECHECK) but $[5/x]e_1 = \langle [5/x]\,T \Rightarrow \{y{:}[5/x]\,T \mid \mathsf{true}\}\rangle^l\, 0 \Rrightarrow \langle \{y{:}[5/x]\,T \mid \mathsf{true}\}, \mathsf{true}, 0\rangle^l$ by EP_RCHECK, not $[5/x]e_2$. Note that the definition of substitution $[e'/x]e$ is a standard one, in which substitution goes down into casts.

Counterexample 2: substitutivity

Let T_2 be a type with a free variable x.

$$\begin{aligned} e_1 &= \langle T_1 \to T_2 \Rightarrow [5/x]\,T_1 \to T_2\rangle^l\, v \\ e_2 &= \lambda y{:}T_1.\, \langle T_2 \Rightarrow [5/x]\,T_2\rangle^l\, (v\, (\langle T_1 \Rightarrow T_1\rangle^l\, y)) \\ e_1' = e_2' &= 5 \end{aligned}$$

Observe that $e_1' \Rrightarrow e_2'$ (by EP_REFL) and $e_1 \Rrightarrow e_2$ (by EP_RFUN). We have $[5/x]e_1 = \langle [5/x]\,T_1 \to T_2 \Rightarrow [5/x]\,T_1 \to T_2\rangle^l\, v \Rrightarrow [5/x]v$ by EP_RREFL, not $[5/x]e_2$.

Counterexample 3: cotermination

$$\begin{aligned} e &= \langle \{x{:}\mathsf{Bool} \mid \mathsf{false}\} \Rightarrow \{x{:}\mathsf{Bool} \mid y\}\rangle^l\, \mathsf{true} \\ e_1 &= 0 = 5 \\ e_2 &= \mathsf{false} \end{aligned}$$

Observe that $e_1 \longrightarrow e_2$ (and so $e_1 \Rrightarrow e_2$, by EP_ROP) and cotermination says that $[e_1/y]e$ terminates at a value iff so does $[e_2/x]e$. Here, by E_CHECK, $[e_1/y]e \longrightarrow \langle \{x{:}\mathsf{Bool} \mid e_1\}, e_1, \mathsf{true}\rangle^l \longrightarrow^* \Uparrow l$ but by E_REFL, $[e_2/x]e \longrightarrow \mathsf{true}$.

Fig. 8. Counterexamples to substitutivity and cotermination of parallel reduction in F_H

Cotermination breaks also because substitutions can affect which reduction rule applies to a cast, which in turn can force us to perform checks under one substitution that aren't performed under another, related one (counterexample 3 in Figure 8).

7.2. F_H 2.0: Greenberg's thesis

In his thesis, Greenberg tried to correct this problem using a fix due to Sekiyama: he takes *common-subexpression reduction* (CSR) as the conversion relation [Greenberg 2013]. We repeat F_H^σ's identical definition of CSR (Figure 4) again here, in Figure 9. As we can see from the definition, CSR is designed to be substitutive (and *is* substitutive). However, cotermination still fails: we can construct untyped terms that don't satisfy cotermination in Greenberg's operational semantics—they look like the term in counterexample 3 (Figure 8). The essential issue is that we can fire E_REFL under one substitution and force a check under another. If the term is ill typed, then we have no way of knowing whether the argument of the cast satisfies its input type—so the check can fail where E_REFL succeeded. Well typed terms don't have this problem, but we need our conversion relation to prove progress and preservation—we can't use arguments about typing in our proof of cotermination. In short, Greenberg's Conjecture 3.2.1 on page 88 is false; it seems that the evaluation relation is defined in such a way that substitutions can't affect which cast reduction rules are chosen.

Conversion $\boxed{\sigma_1 \longrightarrow^* \sigma_2}$ $\boxed{T_1 \equiv T_2}$

$$\sigma_1 \longrightarrow^* \sigma_2 \iff \begin{array}{l}\mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2) \subset \mathsf{TmVar} \wedge \\ \forall x \in \mathsf{dom}(\sigma_1).\ \sigma_1(x) \longrightarrow^* \sigma_2(x)\end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ \text{C\_VAR} \qquad \frac{}{B \equiv B}\ \text{C\_BASE} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \equiv T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \equiv \{x{:}T_2 \mid \sigma_2(e)\}}\ \text{C\_REFINE}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'}\ \text{C\_FUN} \qquad \frac{T \equiv T'}{\forall\alpha.T \equiv \forall\alpha.T'}\ \text{C\_FORALL}$$

$$\frac{T_2 \equiv T_1}{T_1 \equiv T_2}\ \text{C\_SYM} \qquad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}\ \text{C\_TRANS}$$

Fig. 9. Type conversion via common-subexpression reduction

7.3. $\mathrm{F}_{\mathrm{H}}^{\sigma}$

Our calculus, $\mathrm{F}_{\mathrm{H}}^{\sigma}$, can see statically which cast reduction rule is chosen thanks to our definition of substitution (Definition 3.1). In Lemma 5.5, we show that terms related by CSR coterminate at true using $\mathrm{F}_{\mathrm{H}}^{\sigma}$'s substitution semantics; this is enough to prove type soundness and parametricity.

F_{H} tried to use entirely syntactic techniques to achieve type soundness, avoiding the semantic techniques necessary for λ_{H}. But we failed: we need to prove cotermination to get type soundness; our proof amounts to showing that type conversion is a weak bisimulation. Our metatheory is, on the one hand, simpler than that of Greenberg et al. [2010], which needs cotermination *and* semantic type soundness. On the other hand, we must use a nonstandard substitution operation, which is a hassle.

7.4. Discussion

Introducing explicit tagging is an attractive alternative approach. In an explicitly tagged manifest contract system, the only values inhabiting refinement types are tagged as such, e.g., $(v, \{x{:}T \mid e\})$; the operational semantics then manages tags on values, tagging in E_CHECKOK and untagging in E_FORGET. Explicit tagging has several advantages: it clarifies the staging of the operational semantics; it eliminates the need for a T_FORGET rule; it gives value inversion directly (Lemma A.16). Such a semantics would need to get stuck when casts are applied to inappropriately tagged arguments, since typing can't be used in the proof of cotermination.

Finally: what kind of calculus *wouldn't* have cotermination at true for well typed terms? In a nondeterministic language, CSR may make one choice with σ_1 and another with σ_2. Fortunately, F_{H} is deterministic. In a deterministic language, cotermination at true may not hold for CSR if the evaluation relation misuses equalities between terms, e.g., if some rules predicate reduction on subterm equalities which other rules ignore. $\mathrm{F}_{\mathrm{H}}^{\sigma}$ is careful to fix the types in its casts early, delaying substitutions so that they don't affect reduction—the intuition underlying our proof of cotermination.

8. RELATED WORK

We discuss work related to $\mathrm{F}_{\mathrm{H}}^{\sigma}$ in two parts. First, we contrast our work with the untyped contract systems that enforce parametric polymorphism *dynamically*, rather than statically as $\mathrm{F}_{\mathrm{H}}^{\sigma}$ does. We then discuss how $\mathrm{F}_{\mathrm{H}}^{\sigma}$ differs from existing manifest contract calculi in greater detail.

8.1. Dynamically checked polymorphism

The $\mathrm{F}_{\mathrm{H}}^{\sigma}$ 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 [2000]).

Guha et al. [2007] define contracts with polymorphic signatures, maintaining abstraction with sealed "coffers"; they do not prove parametricity. Matthews and Ahmed [2008] prove parametricity for a polymorphic multi-language system with a similar policy. Neis et al. [2009] use dynamic type generation to restore parametricity in the presence of intensional type analysis. $\mathrm{F}_{\mathrm{H}}^{\sigma}$'s contracts are subordinate to the type system, so the parametricity result does not require dynamic type generation. Ahmed et al. [2009] prove parametricity for a gradual typing [Siek and Taha 2006] calculus which enforces polymorphism with a set of global runtime seals. Ahmed et al. [2011] define a polymorphic calculus for gradual typing, using local syntactic "barriers" instead of global seals. The type bindings in that work inspired the delayed substitution in this one. It is probably possible to combine $\mathrm{F}_{\mathrm{H}}^{\sigma}$ with the barrier calculus of Ahmed et al., yielding a polymorphic blame calculus [Wadler and Findler 2009]. How to prove parametricity of such a calculus remains an open question.

8.2. $\mathrm{F}_{\mathrm{H}}^{\sigma}$ and other manifest calculi

Five existing manifest calculi with dependent function types ([Flanagan 2006; Greenberg et al. 2010; Knowles and Flanagan 2010; Ou et al. 2004; Strub et al. 2012]) use subtyping and theorem provers as part of the definition of their type systems. All five of these calculi have complicated metatheory to overcome the same two impediments in the preservation proof: preservation after active checks and after congruence steps in the argument position of applications (see the discussion on T_EXACT and T_CONV in Section 3.3). Ou et al. [2004] restrict refinements and arguments of dependent functions to a conservative approximation of pure terms; they also place strong requirements on their prover. Strub et al. [2012] restrict dependency to values. Knowles and Flanagan [2010] as well as Greenberg et al. [2010] use subtyping to resolve these issues and introduce type semantics to give a firm foundation of subtyping to earlier work [Flanagan 2006]. We consider three systems in more detail: Knowles and Flanagan's λ_{H} (KF); Greenberg et al.'s λ_{H} (which we write here as GPW); and $\mathrm{F}_{\mathrm{H}}^{\sigma}$. (For a comparison of $\mathrm{F}_{\mathrm{H}}^{\sigma}$ to earlier versions of F_{H}, see Section 7). The rest of this subsection addresses the differences between KF, GPW, and $\mathrm{F}_{\mathrm{H}}^{\sigma}$. More concretely, we discuss (1) how subtyping in KF and GPW is defined and used, (2) why denotational semantics is needed, and (3) how $\mathrm{F}_{\mathrm{H}}^{\sigma}$ avoids complications introduced by subtyping and denotational semantics.

KF and GPW use a rule like the following for refinement subtyping:[8]

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

(Note that B is a base type, while $\mathrm{F}_{\mathrm{H}}^{\sigma}$ allows refinements of any type.) Combined with a "constants get most specific types" requirement—for example, assigning n the type $\{x{:}\mathsf{Int} \mid x = n\}$—subtyping allows n to be typed at any predicate contract it satisfies, resolving the first issue of showing preservation after active checks. For example, if $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^{l}\, n \longrightarrow^* n$, then n can be given type $\{x{:}\mathsf{Int} \mid x > 0\}$ (because $n > 0 \longrightarrow^* \mathsf{true}$). Subtyping is used in KF and GPW also to address the second issue,

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

which amounts to showing the equivalence of types $[e/x]T$ and $[e'/x]T$ when $e \longrightarrow e'$. Both KF and GPW relate reduction and subtyping, showing that types that reduce to each other are mutual subtypes. KF use full beta reduction throughout their system. GPW use call-by-value reduction in their operational semantics, showing that parallel reducing types are mutual subtypes, separately relating CBV and parallel reduction. Once these two difficulties are resolved, both preservation proofs are standard, given appropriate subtyping inversion lemmas.

So much for motivation for subtyping. Now, why do KF and GPW need a denotational semantics for types? 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, i.e., substitution of "well-typed" closed values for term variables, is in a negative position. Where do well-formed closing substitutions come from? We cannot use syntactic typing, as this would be ill-defined: term typing requires subtyping via subsumption; subtyping requires well-formed closing substitutions in a negative position via the refinement case; but well-formed closing substitutions require typing. We need another source of "well-typed" values: hence, the denotations of types. 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:

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

in particular

$$\emptyset \vdash e : T \text{ implies } e \in [\![T]\!].$$

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 somewhat unsatisfying. We set out to prove syntactic type soundness and ended up proving semantic type soundness along the way. While not a serious burden for a language as small as λ_{H}, having to use semantic techniques throughout makes adding some features—polymorphism, state and other effects, concurrency—difficult. For example, a semantic proof of type soundness for $\mathrm{F}_{\mathrm{H}}^{\sigma}$ would be very close to a proof of parametricity—must we prove parametricity while proving type soundness?

$\mathrm{F}_{\mathrm{H}}^{\sigma}$ solves the problem by avoiding subtyping—which is what forced the presence of closing substitutions and denotational semantics in the first place—and introducing T_EXACT, T_CONV, and convertibility $\equiv$ instead. We would like to note that T_EXACT

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

needs some care to avoid vicious circularity: it is crucial to stipulate v and $\{x{:}T \mid e\}$ be closed. If we "bit the bullet" and allowed non-empty contexts there, then we would need to apply a closing substitution to $[v/x]e$ before checking if it reduces to true but it would lead to the same circularity as subtyping we discussed above. As for T_CONV and convertibility, convertibility is much simpler than GPW and Belo et al. [2011]. It doesn't, unfortunately, completely simplify the proof: we must prove that our conversion relation is a weak bisimulation to establish *cotermination* (Lemma 5.5) before proving type soundness.

Another consequence of dropping subtyping is that the type system of F_H^σ is not a superset of that of λ_H.[9] In particular, λ_H builds in subsumption, while F_H^σ only has a subsumption principle *post facto*, per the proofs in prior version of F_H [Belo et al. 2011; Greenberg 2013]. We can, however, take a λ_H typing derivation and eliminate every occurrence of subsumption: by the upcast lemma, the two programs are equivalent, even if one of them is not well typed. That is, we have taken subsumption out of the type system and proved subsumption safe as an optimization—and, in doing so, greatly simplified the type system.

9. CONCLUSION

F_H^σ offers a simpler approach to combining parametric polymorphism and manifest contracts. When we say "parametrically" polymorphic, we mean in particular that the relation R used to related terms at type variables in the logical relation is a *parameter* of the logical relation, which admits any instantiation of R.[10] We offer the first conjecture-free, completely correct operational semantics for *general* refinements, where refinements can apply to any type, not just base types.

We hope to extend F_H with barriers for dynamically checked polymorphism [Ahmed et al. 2011], and with and state. (Though we acknowledge that state is a difficult open problem.) We also hope that F_H's operational semantics and (relatively) simple type system will help developers implement contracts. With the introduction of abstract types, there is room to draw connections between the client/server blame from Section 1 and Findler and Felleisen-style client/server blame. Finally, we are curious to see what we can do with a contract language with the reasoning principles derivable from relational parametricity.

Acknowledgments

João Felipe Belo and Benjamin Pierce helped do the original work on F_H. Stephanie Weirich provided many insights throughout. Jianzhou Zhao's help with parametricity was invaluable; and a conversation about parametricity with Amal Ahmed and Stephanie Weirich was particularly illuminating. A conversation with Cătălin Hriţcu led to the functional interpretation of cast semantics.

REFERENCES

AHMED, A., FINDLER, R. B., MATTHEWS, J., AND WADLER, P. 2009. Blame for all. In *Workshop on Script-to-Program Evolution (STOP)*.

AHMED, A., FINDLER, R. B., SIEK, J., AND WADLER, P. 2011. Blame for all. In *Principles of Programming Languages (POPL)*.

BELO, J. F., GREENBERG, M., IGARASHI, A., AND PIERCE, B. C. 2011. Polymorphic contracts. In *European Symposium on Programming (ESOP)*.

BENZAKEN, V., CASTAGNA, G., AND FRISCH, A. 2003. CDuce: an XML-centric general-purpose language. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 51–63.

BIERMAN, G. M., GORDON, A. D., HRIŢCU, C., AND LANGWORTHY, D. 2010. Semantic subtyping with an SMT solver. In *International Conference on Functional Programming (ICFP)*.

[9]The F_H^σ operational semantics is essentially a superset of λ_H from Greenberg et al. [2010], barring some slight differences in the function cast decomposition rule.

[10]Earlier versions [Belo et al. 2011] only admit relations that respect parallel reduction, but that restriction has been relaxed.

BOHANNON, A., FOSTER, J. N., PIERCE, B. C., PILKIEWICZ, A., AND SCHMITT, A. 2008. Boomerang: resourceful lenses for string data. In *Principles of Programming Languages (POPL)*.

FELLEISEN, M. AND HIEB, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science (TCS) 103,* 2, 235–271.

FINDLER, R. B. AND FELLEISEN, M. 2002. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*.

FLANAGAN, C. 2006. Hybrid type checking. In *Principles of Programming Languages (POPL)*.

FLATT, M. AND PLT. 2010. Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. `http://racket-lang.org/tr1/`.

GREENBERG, M. 2013. Manifest contracts. Ph.D. thesis, University of Pennsylvania.

GREENBERG, M., PIERCE, B. C., AND WEIRICH, S. 2010. Contracts made manifest. In *Principles of Programming Languages (POPL)*.

GRONSKI, J. AND FLANAGAN, C. 2007. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*.

GUHA, A., MATTHEWS, J., FINDLER, R. B., AND KRISHNAMURTHI, S. 2007. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*.

HARPER, R., HONSELL, F., AND PLOTKIN, G. 1993. A framework for defining logics. *J. ACM 40,* 1, 143–184.

KNOWLES, K. AND FLANAGAN, C. 2010. Hybrid type checking. *ACM Trans. Program. Lang. Syst. 32*, 6:1–6:34.

MATTHEWS, J. AND AHMED, A. 2008. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*.

NEIS, G., DREYER, D., AND ROSSBERG, A. 2009. Non-parametric parametricity. In *International Conference on Functional Programming (ICFP)*.

OU, X., TAN, G., MANDELBAUM, Y., AND WALKER, D. 2004. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science (TCS)*.

PIERCE, B. AND SUMII, E. 2000. Relating cryptography and polymorphism.

PITTS, A. M. 2005a. Typed operational reasoning. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. The MIT Press, Chapter 7, 245–289.

PITTS, A. M. 2005b. Typed operational reasoning. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. The MIT Press, Chapter 7.

PLT 2014. Racket contract system.

SEKIYAMA, T. AND IGARASHI, A. 2014. Logical relations for a manifest calculus, fixed. In preparation for submission.

SEKIYAMA, T., NISHIDA, Y., AND IGARASHI, A. 2015. Manifest contracts for datatypes. In *Principles of Programming Languages (POPL)*.

SIEK, J. G. AND TAHA, W. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*.

STRUB, P.-Y., SWAMY, N., FOURNET, C., AND CHEN, J. 2012. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *Principles of Programming Languages (POPL)*. ACM.

TABUCHI, N., SUMII, E., AND YONEZAWA, A. 2003. Regular expression types for strings in a text processing language. *Electronic Notes in Theoretical Computer Science*. International Workshop in Types in Programming.

WADLER, P. 1989. Theorems for free! In *Conference on Functional Programming and Computer Architecture (FPCA)*.

WADLER, P. AND FINDLER, R. B. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation 115*, 38–94.

A. PROOFS

A.1. Cotermination

We first start with proving standard properties about free variables and substitution because they are non-standard and slightly tricky.

A.1 Lemma: Let σ be a substitution.

(1) For any term e, $\mathrm{FV}(\sigma(e)) = (\mathrm{FV}(e) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma \mid \mathrm{AFV}(e))$.
(2) For any type T, $\mathrm{FV}(\sigma(T)) = (\mathrm{FV}(T) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma \mid \mathrm{AFV}(T))$.

PROOF. By structural induction on e and T. □

A.2 Lemma: Let σ be a substitution.

(1) For any term e, $\mathrm{FTV}(\sigma(e)) = (\mathrm{FTV}(e) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FTV}(\sigma \mid \mathrm{AFV}(e))$.
(2) For any type T, $\mathrm{FTV}(\sigma(T)) = (\mathrm{FTV}(T) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FTV}(\sigma \mid \mathrm{AFV}(T))$.

PROOF. By structural induction on e and T. □

A.3 Lemma: Let σ be a substitution.

(1) If $\mathrm{AFV}(e) \cap \mathrm{dom}(\sigma) = \emptyset$, then $\sigma(e) = e$.
(2) If $\mathrm{AFV}(T) \cap \mathrm{dom}(\sigma) = \emptyset$, then $\sigma(T) = T$.

PROOF. By structural induction on e and T. □

A.4 Lemma: Let σ_1 and σ_2 be substitutions. Suppose that $\mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset$ and $\mathrm{AFV}(\sigma_2) \cap \mathrm{dom}(\sigma_1) = \emptyset$.

(1) For any term e, $\sigma_2(\sigma_1(e)) = (\sigma_2(\sigma_1))(\sigma_2(e))$.
(2) For any type T, $\sigma_2(\sigma_1(T)) = (\sigma_2(\sigma_1))(\sigma_2(T))$.

PROOF. By structural induction on e and T with Lemma A.3. □

A.5 Lemma [Determinism (Lemma 5.1)]: If $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.

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

A.6 Lemma: Suppose that e_1 and e_2 are closed terms and that e'_1, $[e_1/x]e'_2$ and $[e_2/x]e'_2$ are values. If $[e_1/x](e'_1\, e'_2) \longrightarrow e$, then $[e_2/x](e'_1\, e'_2) \longrightarrow [e_2/x]e'$ for some e' such that $e = [e_1/x]e'$.

PROOF. By case analysis on e'_1. Note that e'_1 takes the form of either lambda abstraction or cast since the application term $[e_1/x](e'_1\, e'_2)$ takes a step. We give just two emblematic cases: E_FUN and E_PRECHECK.

$\underline{e'_1 = \langle y{:}T_{11} \rightarrow T_{12} \Rightarrow y{:}T_{21} \rightarrow T_{22}\rangle^l_\sigma \text{ where } y{:}T_{11} \rightarrow T_{12} \neq y{:}T_{21} \rightarrow T_{22}}$: Without loss of generality, we can suppose that y and variables of $\mathrm{dom}(\sigma)$ are fresh. Let z be a fresh variable and $i, j \in \{1, 2\}$. Moreover, let σ_i be

$$[e_i/x]\sigma \uplus ([e_i/x] \mid (\mathrm{AFV}(y{:}T_{11} \rightarrow T_{12}) \cup \mathrm{AFV}(y{:}T_{21} \rightarrow T_{22})) \setminus \mathrm{dom}(\sigma))$$

and σ_{ij} be $\sigma_i \mid \mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})$. Then, $[e_i/x]e'_1 = \langle y{:}T_{11} \rightarrow T_{12} \Rightarrow y{:}T_{21} \rightarrow T_{22}\rangle^l_{\sigma_i}$ and, by E_REDUCE/E_FUN, $[e_i/x](e'_1\, e'_2) \longrightarrow e''_i$ where

$$e''_i = \lambda y{:}\sigma_i(T_{21}).\ \mathsf{let}\ z : \sigma_i(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_{i1}}\ y\ \mathsf{in}\ \langle [z/y]T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_{i2}}\ ([e_i/x]e'_2\ z).$$

Here, let $\sigma'_j = \sigma \mid \mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})$ and e' be

$$\lambda y{:}\sigma(T_{21}).\ \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^{l}_{\sigma'_1}\ y\ \mathsf{in}\ \langle [z/y]T_{12} \Rightarrow T_{22}\rangle^{l}_{\sigma'_2}\ (e'_2\ z)$$

for some fresh variable z.
We show $[e_i/x]e' = e''_i$. By Lemma A.4, $[e_i/x]\sigma(T_{21}) = ([e_i/x]\sigma)([e_i/x]T_{21}) = \sigma_i(T_{21})$ and, similarly, $[e_i/x]\sigma(T_{11}) = \sigma_i(T_{11})$. Also, letting $S_j = \mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})$,

$$\begin{array}{ll} & [e_i/x]\sigma'_j \uplus ([e_i/x] \mid S_j \setminus \mathrm{dom}(\sigma'_j)) \\ = & [e_i/x]\sigma'_j \uplus ([e_i/x] \mid S_j \setminus \mathrm{dom}(\sigma)) \qquad \text{(because } S_j \setminus \mathrm{dom}(\sigma'_j) = S_j \setminus \mathrm{dom}(\sigma)) \\ = & ([e_i/x]\sigma \mid S_j) \uplus ([e_i/x] \mid S_j \setminus \mathrm{dom}(\sigma)) \\ = & (\sigma_i \mid \mathrm{dom}(\sigma) \cap S_j) \uplus ([e_i/x] \mid S_j \setminus \mathrm{dom}(\sigma)) \\ = & (\sigma_{ij} \mid \mathrm{dom}(\sigma)) \uplus ([e_i/x] \mid S_j \setminus \mathrm{dom}(\sigma)) \\ = & \sigma_{ij}. \end{array}$$

The last equation is derived from the fact that

$$\begin{array}{rcl} x \in \mathsf{dom}(\sigma_{ij}) & \iff & x \in S_j \cap \mathrm{dom}(\sigma_i) \\ & \iff & x \in S_j \cap ((\mathrm{AFV}(y{:}T_{11} \to T_{12}) \cup \mathrm{AFV}(y{:}T_{21} \to T_{22})) \setminus \mathrm{dom}(\sigma)) \\ & \iff & x \in (S_j \cap (\mathrm{AFV}(y{:}T_{11} \to T_{12}) \cup \mathrm{AFV}(y{:}T_{21} \to T_{22}))) \setminus \mathrm{dom}(\sigma) \\ & \iff & x \in S_j \setminus \mathrm{dom}(\sigma). \end{array}$$

$\underline{e'_1 = \langle T_1 \Rightarrow \{y{:}T_2 \mid e\}\rangle^{l}_{\sigma}}$: Here $T_1 \neq \{y{:}T_2 \mid e\}$ and $T_1 \neq T_2$ and $T_1 \neq \{z{:}T' \mid e'\}$ for any z, T' and e'. Let $i \in \{1, 2\}$ and

$$\begin{array}{rcl} \sigma_i & = & [e_i/x]\sigma \uplus ([e_i/x] \mid (\mathrm{AFV}(T_1) \cup \mathrm{AFV}(\{y{:}T_2 \mid e\})) \setminus \mathrm{dom}(\sigma)) \\ \sigma_{i1} & = & \sigma_i \mid \mathrm{AFV}(\{y{:}T_2 \mid e\}) \\ \sigma_{i2} & = & \sigma_i \mid \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2). \end{array}$$

Then, by E_REDUCE/E_PRECHECK, $[e_i/x](e'_1\ e'_2) \longrightarrow e''_i$ where

$$e''_i = \langle T_2 \Rightarrow \{y{:}T_2 \mid e\}\rangle^{l}_{\sigma_{i1}}\ (\langle T_1 \Rightarrow T_2\rangle^{l}_{\sigma_{i2}}\ [e_i/x]e'_2).$$

Letting

$$\begin{array}{rcl} \sigma'_1 & = & \sigma \mid \mathrm{AFV}(\{y{:}T_2 \mid e\}) \\ \sigma'_2 & = & \sigma \mid \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2) \\ e' & = & \langle T_2 \Rightarrow \{y{:}T_2 \mid e\}\rangle^{l}_{\sigma'_1}\ (\langle T_1 \Rightarrow T_2\rangle^{l}_{\sigma'_2}\ e'_2), \end{array}$$

it suffices to show that $[e_i/x]e' = e''_i$. We can show that $[e_i/x]\sigma'_1 \uplus ([e_i/x] \mid \mathrm{AFV}(\{y{:}T_2 \mid e\}) \setminus \mathrm{dom}(\sigma'_1)) = \sigma_{i1}$ and and $[e_i/x]\sigma'_2 \uplus ([e_i/x] \mid (\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma'_2)) = \sigma_{i2}$ similarly to the above, and so we finish.

□

A.7 Lemma: Suppose that $e_1 \longrightarrow e_2$ and that $[e_1/x]e'_1$, $[e_1/x]e'_2$ and $[e_2/x]e'_2$ are values.

(1) If $[e_1/x](e'_1\ e'_2) \longrightarrow e$, then $[e_2/x](e'_1\ e'_2) \longrightarrow [e_2/x]e'$ for some e' such that $e = [e_1/x]e'$.
(2) If $[e_2/x](e'_1\ e'_2) \longrightarrow e$, then $[e_1/x](e'_1\ e'_2) \longrightarrow [e_1/x]e'$ for some e' such that $e = [e_2/x]e'$.

PROOF. Since $[e_1/x]e'_1$ is a value, and e_1 is not a value from $e_1 \longrightarrow e_2$, we have e'_1 is not a variable, and thus e'_1 is a value from the assumption that so is $[e_1/x]e'_1$. Since evaluation relation is defined over closed terms, we finish by Lemma A.6. □

A.8 Lemma: Suppose that e_1 and e_2 are closed terms and that e is a value. If $[e_1/x](e\ T) \longrightarrow e'$, then $[e_2/x](e\ T) \longrightarrow [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.

PROOF. Since the type application term $[e_1/x](e\ T)$ takes a step, e takes the form of type abstraction. Let $e = \Lambda\alpha.\ e'$. Without loss of generality, we can suppose that α is fresh. Let $i \in \{1, 2\}$. By E_REDUCE/E_TBETA, $[e_i/x](e\ T) \longrightarrow [[e_i/x]T/\alpha][e_i/x]e'$. Since e_i is closed, we have $[[e_i/x]T/\alpha][e_i/x]e' = [e_i/x][T/\alpha]e'$ by Lemma A.4 (1), and thus we finish. □

A.9 Lemma: Suppose that $e_1 \longrightarrow e_2$ and that $[e_1/x]e$ is a value.

(1) If $[e_1/x](e\ T) \longrightarrow e'$, then $[e_2/x](e\ T) \longrightarrow [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.
(2) If $[e_2/x](e\ T) \longrightarrow e'$, then $[e_1/x](e\ T) \longrightarrow [e_1/x]e''$ for some e'' such that $e' = [e_2/x]e''$.

PROOF. By Lemma A.8 because it is found that e is a value and that e_1 and e_2 are closed terms. □

A.10 Lemma: Suppose that $e_1 \longrightarrow e_2$. If $[e_1/x]e \longrightarrow e'$, then $[e_2/x]e \longrightarrow^* [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.

PROOF. By structural induction on e. Note that e_1 is not a value from $e_1 \longrightarrow e_2$.

$\underline{e = x}$: Since $[e_1/x]e = e_1$ and $[e_2/x]e = e_2$, we finish by Lemma A.3 when letting $e'' = e_2$ because e_2 is closed.
$\underline{e = v,\ y \text{ where } x \neq y \text{ or } \Uparrow l}$: Contradiction from $[e_1/x]e \longrightarrow e'$.
$\underline{e = \mathrm{op}\,(e'_1, .., e'_n)}$: If all terms $[e_1/x]e'_i$ are values, then they are constants since $[e_1/x]\,\mathrm{op}\,(e'_1, ..., e'_n)$ takes a step. Since e_1 is not a value, $e'_i = k_i$ for some k_i. Thus, $[e_1/x]e = [e_2/x]e = \mathrm{op}\,(k_1, ..., k_n)$ and so we finish.
Otherwise, we suppose that some $[e_1/x]e'_i$ is not a value and all terms to the left of $[e_1/x]e'_i$ are values. From that, we can show that all terms to the left of $[e_2/x]e'_i$ are values since e_1 is not a value. If $[e_1/x]e'_i$ gets stuck, then contradiction because $[e_1/x]e$ takes a step. If $[e_1/x]e'_i \longrightarrow e''$, then, by the IH, $[e_2/x]e'_i \longrightarrow^* [e_2/x]e''_i$ for some e''_i such that $e'' = [e_1/x]e''_i$. Thus, we finish by E_COMPAT. Otherwise, if $[e_1/x]e'_i = \Uparrow l$, then $[e_2/x]e'_i = \Uparrow l$ because $e'_i = \Uparrow l$ by $e_1 \neq \Uparrow l$, which follows from $e_1 \longrightarrow e_2$. Thus, we finish by E_BLAME.
$\underline{e = e'_1\ e'_2}$: We can show the case where either $[e_1/x]e'_1$ or $[e_1/x]e'_2$ is not a value similarly to the above. Otherwise, if they are values, we can find that so are $[e_2/x]e'_1$ and $[e_2/x]e'_2$, and thus we finish by Lemma A.7 (1).
$\underline{e = e'_1\ T_2}$: Similarly to the case of function application, with Lemma A.9 (1).
$\underline{e = \langle\{y{:}T \mid e'_1\}, e'_2, v\rangle^l}$: Similarly to the above.

□

A.11 Lemma: If $e_1 \longrightarrow e_2$, and $[e_2/x]e$ is a value, then there exists some e' such that

— $[e_1/x]e \longrightarrow^* [e_1/x]e'$,
— $[e_1/x]e'$ is a value, and
— $[e_2/x]e = [e_2/x]e'$.

PROOF. By case analysis on e. □

A.12 Lemma: If $e_1 \longrightarrow e_2$, and $[e_2/x]e = \Uparrow l$, then $[e_1/x]e \longrightarrow^* \Uparrow l$.

PROOF. By case analysis on e. □

A.13 Lemma [Cotermination]: (Lemma 5.3)
Suppose that $e_1 \longrightarrow e_2$. If $[e_2/x]e \longrightarrow e'$, then $[e_1/x]e \longrightarrow^* [e_1/x]e''$ for some e'' such that $e' = [e_2/x]e''$.

PROOF. By structural induction on e.

$\underline{e = x}$: Since $[e_1/x]e = e_1$ and $[e_2/x]e = e_2$, we finish by Lemma A.3 when letting $e'' = e'$.
$\underline{e = v,\ y \text{ where } x \neq y \text{ or } \Uparrow l}$: Contradiction from $[e_2/x]e \longrightarrow e'$.
$\underline{e = \mathrm{op}\,(e'_1, .., e'_n)}$: If all terms $[e_2/x]e'_i$ are values, then they are constants since $[e_2/x]\,\mathrm{op}\,(e'_1, ..., e'_n)$ takes a step. By Lemma A.11, $[e_1/x]\,\mathrm{op}\,(e'_1, ..., e'_n) \longrightarrow^* [e_1/x]\,\mathrm{op}\,(e''_1, ..., e''_2)$ for some $e''_1, ..., e''_n$ such that $[e_2/x]\,\mathrm{op}\,(e'_1, ..., e'_n) = [e_2/x]\,\mathrm{op}\,(e''_1, ..., e''_n)$. Since e_1 is not a value from $e_1 \longrightarrow e_2$, $e''_i = k_i$ for some k_i. Thus, we finish.
Otherwise, we suppose that some $[e_2/x]e'_i$ is not a value and all terms to the left of $[e_2/x]e'_i$ are values. By Lemma A.11, each term $[e_1/x]e'_j$ to the left of $[e_1/x]e'_i$ evaluates to a value $[e_1/x]e''_j$ for some e''_j such that $[e_2/x]e'_j = [e_2/x]e''_j$. If $[e_2/x]e'_i$ gets stuck, then contradiction because $[e_2/x]e$ takes a step. If $[e_2/x]e'_i = \Uparrow l$, then $[e_1/x]e'_i \longrightarrow^* \Uparrow l$ by Lemma A.12. Thus, we finish by E_BLAME. Otherwise, if $[e_2/x]e'_i \longrightarrow e''$, then we finish by the IH and E_COMPAT.
$\underline{e = e'_1\ e'_2}$: We can show the case where either $[e_2/x]e'_1$ or $[e_2/x]e'_2$ is not a value similarly to the above. Otherwise, if they are values, we can find, by Lemma A.11, that $[e_1/x]e'_1$ and $[e_1/x]e'_2$ evaluates to values $[e_1/x]e''_1$ and $[e_1/x]e''_2$ for some e''_1 and e''_2 such that $[e_2/x]e'_1 = [e_2/x]e''_1$ and $[e_2/x]e'_2 = [e_2/x]e''_2$, respectively. Then, we finish by Lemma A.7 (2).
$\underline{e = e'_1\ T_2}$: Similarly to the case of function application, with Lemma A.9 (2).
$\underline{e = \langle \{y{:}T \mid e'_1\}, e'_2, v\rangle^l}$: Similarly to the above.

□

A.14 Lemma [Cotermination]: (Lemma 5.4)

(1) Suppose that $e_1 \longrightarrow e_2$.
  (a) If $[e_1/x]e \longrightarrow^* \mathsf{true}$, then $[e_2/x]e \longrightarrow^* \mathsf{true}$.
  (b) If $[e_2/x]e \longrightarrow^* \mathsf{true}$, then $[e_1/x]e \longrightarrow^* \mathsf{true}$.
(2) Suppose that $e_1 \longrightarrow^* e_2$.
  (a) If $[e_1/x]e \longrightarrow^* \mathsf{true}$, then $[e_2/x]e \longrightarrow^* \mathsf{true}$.
  (b) If $[e_2/x]e \longrightarrow^* \mathsf{true}$, then $[e_1/x]e \longrightarrow^* \mathsf{true}$.

PROOF.

(1) By induction on the number of evaluation steps of $[e_1/x]e$ and $[e_2/x]e$ with Lemma A.10 and Lemmas A.11 and A.13, respectively.
(2) By induction on the number of evaluation steps of e_1 with the first case.

□

A.15 Lemma [Cotermination of refinement types (Lemma 5.6)]: If $\{x{:}T_1 \mid e_1\} \equiv \{x{:}T_2 \mid e_2\}$ then $T_1 \equiv T_2$ and $[v/x]e_1 \longrightarrow^* \mathsf{true}$ iff $[v/x]e_2 \longrightarrow^* \mathsf{true}$, for any closed value v.

PROOF. By induction on the equivalence. There are three cases.
(C_REFINE): We have $T_1 \equiv T_2$ by assumption. We know that $e_1 = \sigma_1(e)$ and $e_2 = \sigma_2(e)$ for $\sigma_1 \longrightarrow^* \sigma_2$. It is trivially true that $v \longrightarrow^* v$, so $[v/x]\sigma_1 \longrightarrow^* [v/x]\sigma_2$. By cotermination (Lemma 5.5), we know that $[v/x]\sigma_1(e) \longrightarrow^* \mathsf{true}$ iff $[v/x]\sigma_2(e) \longrightarrow^* \mathsf{true}$.
(C_SYM): By the IH.
(C_TRANS): By the IHs and transitivity of $\equiv$ and cotermination. □

A.2. Type soundness

A.16 Lemma [Value inversion (Lemma 5.7)]: If $\emptyset \vdash v : T$ and $\mathrm{unref}_n(T) = \{x{:}T_n \mid e_n\}$ then $[v/x]e_n \longrightarrow^* \mathsf{true}$.

PROOF. By induction on the height of the typing derivation; we list all the cases that could type values.
(T_CONST): By assumption of valid typing of constants.
(T_ABS): Contradictory—the type is wrong.
(T_TABS): Contradictory—the type is wrong.
(T_CAST): Contradictory—the type is wrong.
(T_CONV): By applying Lemma A.15 on the stack of refinements on T.
(T_FORGET): By the IH on $\emptyset \vdash v : \{x{:}T \mid e\}$, adjusting each of the n down by one to cover the stack of refinements on T.
(T_EXACT): By assumption for the outermost refinement; by the IH on $\emptyset \vdash v : T$ for the rest. □

A.17 Lemma [Reflexivity of conversion]:
$T \equiv T$ for all T.

PROOF. By induction on T. □

A.18 Lemma [Like-type arrow conversion]: If $x{:}T_{11} \rightarrow T_{12} \equiv T$ then $T = x{:}T_{21} \rightarrow T_{22}$.

PROOF. By induction on the conversion relation. Only C_FUN applies, and C_SYM and C_TRANS are resolved by the IH. □

A.19 Lemma [Conversion arrow inversion]: If $x{:}T_{11} \rightarrow T_{12} \equiv x{:}T_{21} \rightarrow T_{22}$ then $T_{11} \equiv T_{21}$ and $T_{12} \equiv T_{22}$.

PROOF. By induction on the conversion derivation, using Lemma A.18. □

A.20 Lemma [Like-type forall conversion]: If $\forall\alpha.T_1 \equiv T$ then $T = \forall\alpha.T_2$.

PROOF. By induction on the conversion relation. Only C_FORALL applies, and C_SYM and C_TRANS are resolved by the IH. □

A.21 Lemma [Conversion forall inversion]: If $\forall\alpha.T_1 \equiv \forall\alpha.T_2$ then $T_1 \equiv T_2$.

PROOF. By induction on the conversion derivation, using Lemma A.20. □

A.22 Lemma [Term substitutivity of conversion (Lemma 5.8)]:
If $T_1 \equiv T_2$ and $e_1 \longrightarrow^* e_2$ then $[e_1/x]T_1 \equiv [e_2/x]T_2$.

PROOF. By induction on $T_1 \equiv T_2$.
(C_VAR): By C_VAR.
(C_BASE): By C_BASE.
(C_REFINE): $T_1 = \{y{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{y{:}T_2' \mid \sigma_2(e)\}$ such that $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$. By the IH on $T_1' \equiv T_2'$, we know that $[e_1/x]T_1' \equiv [e_2/x]T_2'$. Since $e_1 \longrightarrow^* e_2$, we know that $\sigma_1 \uplus [e_1/x] \longrightarrow^* \sigma_2 \uplus [e_2/x]$, and we are done by C_REFINE.
(C_FUN): By the IHs and C_FUN.
(C_FORALL): By the IH and C_FORALL.
(C_TRANS): By the IHs and C_TRANS.
(C_SYM): By the IHs and C_SYM. □

A.23 Lemma [Type substitutivity of conversion (Lemma 5.9)]:
If $T_1 \equiv T_2$ then $[T/\alpha]T_1 \equiv [T/\alpha]T_2$.

PROOF. By induction on $T_1 \equiv T_2$.

(C_VAR): If $T_1 = T_2 = \alpha$, then by reflexivity (Lemma A.17). Otherwise, by C_VAR.
(C_BASE): By C_BASE.
(C_REFINE): $T_1 = \{y{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{y{:}T_2' \mid \sigma_2(e)\}$ such that $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$. By the IH on $T_1' \equiv T_2'$, we know that $[T/\alpha]\,T_1' \equiv [T/\alpha]\,T_2'$. Since $[T/\alpha]\sigma_1 = \sigma_1$ and $[T/\alpha]\sigma_2 = \sigma_2$, so we are done by C_REFINE.
(C_FUN): By the IHs and C_FUN.
(C_FORALL): By the IH and C_FORALL, possibly varying the bound variable name.
(C_SYM): By the IH and C_SYM.
(C_TRANS): By the IHs and C_TRANS. □

A.24 Lemma [Term weakening (Lemma 5.10)]: 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'. The only interesting case is for terms where a runtime rule applies:
(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. □

A.25 Lemma [Type weakening (Lemma 5.11)]: 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'. The proof is similar to term weakening, Lemma A.24. □

A.26 Lemma [Compatibility is symmetric]: $T_1 \parallel T_2$ iff $T_2 \parallel T_1$.

PROOF. By induction on $T_1 \parallel T_2$.
(SIM_VAR): By SIM_VAR.
(SIM_BASE): By SIM_BASE.
(SIM_REFINEL): By SIM_REFINER and the IH.
(SIM_REFINER): By SIM_REFINEL and the IH.
(SIM_FUN): By SIM_FUN and the IHs.
(SIM_FORALL): By the IH and SIM_FORALL. □

A.27 Lemma [Substitution preserves compatibility]:
If $T_1 \parallel T_2$, then $[e/x]\,T_1 \parallel T_2$.

PROOF. By induction on the compatibility relation.
(SIM_VAR): By SIM_VAR.
(SIM_BASE): By SIM_BASE.
(SIM_REFINEL): By SIM_REFINEL and the IH.
(SIM_REFINER): By SIM_REFINER and the IH.
(SIM_FUN): By SIM_FUN and the IHs.
(SIM_FORALL): By SIM_FORALL and the IH. □

A.28 Lemma [Term substitution (Lemma 5.12)]: If $\Gamma \vdash e' : T'$, then

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

PROOF. By induction on e, T, and Γ'. In the first two clauses, we are careful to leave Γ' as long as it is well formed. $\square$

A.29 Lemma [Type substitution preserves compatibility]: If $T_1 \parallel T_2$ then $[T'/\alpha]T_1 \parallel [T'/\alpha]T_2$.

PROOF. By induction on the compatibility relation.
(SIM_VAR): By SIM_VAR or reflexivity of the compatibility (proved easily).
(SIM_BASE): By SIM_BASE.
(SIM_REFINEL): By SIM_REFINEL and the IH.
(SIM_REFINER): By SIM_REFINER and the IH.
(SIM_FUN): By SIM_FUN and the IHs.
(SIM_FORALL): By SIM_FORALL and the IH.
$\square$

A.30 Lemma [Identity type substitution on one side preserves compatibility]: If $T_1 \parallel T_2$ then $[\alpha/\alpha]T_1 \parallel T_2$.

PROOF. Similar to Lemma A.29. $\square$

A.31 Lemma [Type substitution]: If $\Gamma \vdash T'$ then

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

PROOF. By induction on e, T, and Γ'. $\square$

A.32 Lemma [Lambda inversion (Lemma 5.14)]: 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) $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T)$.

PROOF. By induction on the typing derivation. Cases not mentioned only apply to terms which are not lambdas.
(T_ABS): By inversion, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. We find conversion immediately by reflexivity (Lemma A.17), since $\mathrm{unref}(T) = T = x{:}T_1 \to T_2$.
(T_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 $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$ where, $\mathrm{unref}(T') \equiv x{:}T_1 \to T_2$. By weakening, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $T' \equiv T$, we have $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T') \equiv \mathrm{unref}(T)$ by C_TRANS.
(T_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, $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$, where $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T')$. By weakening, $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $\mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$, we have the conversion by C_TRANS): $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$.
(T_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 $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$, where $x{:}T_1 \to T_2 \equiv \mathrm{unref}(\{x{:}T \mid e\})$. By weakening, $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$, we have by C_TRANS that $x{:}T_1 \to T_{12} \equiv \mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$. $\square$

A.33 Lemma [Cast inversion]: If $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : T$, then

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

(5) $\mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)$.

PROOF. By induction on the typing derivation, as for A.32. □

A.34 Lemma [Type abstraction inversion]: If $\Gamma \vdash \Lambda\alpha.\ e : T$, then

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

PROOF. By induction on the typing derivation, as for A.32. □

A.35 Lemma [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 $T_1 \equiv T_2$. □

A.36 Lemma [Canonical forms (Lemma 5.15)]: If $\emptyset \vdash v : T$, then:

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

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 (Lemma A.17).
(T_APP): Contradictory: $e_1\ e_2$ is not a value.
(T_TABS): $\emptyset \vdash \Lambda\alpha.\ e : \forall\alpha.T$; we are in case 3. It is immediate that $v = \Lambda\alpha.\ e$, and conversion is by reflexivity (Lemma A.17).
(T_TAPP): Contradictory: $e\ T$ is not a value.
(T_CAST): $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle_\sigma^l : \_{:}\sigma(T_1) \to \sigma(T_2)$; we are in case 2b. It is immediate that $v = \langle T_1 \Rightarrow T_2 \rangle_\sigma^l$. Conversion is by reflexivity (Lemma A.17).
(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 C_TRANS.

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 $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ (Lemma A.35) and so $\mathrm{unref}(T') = x{:}T_1' \to T_2'$ for some T_1' and T_2' such that $T_1' \equiv T_1$ (Lemma A.19); by C_TRANS, we have $T_1'' \equiv T_1'$.
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 $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ (Lemma A.35) and so $\mathrm{unref}(T') = x{:}T_1'' \to T_2''$ for some T_1'' and T_2'' such that $T_1'' \equiv T_1$ and $T_2'' \equiv T_2$ (Lemma A.19); by C_TRANS, we have $T_1' \equiv T_1''$ and $T_2' \equiv T_2''$ as required.
Case 3: $\mathrm{unref}(T) = \forall\alpha.T_0$ and v is $\Lambda\alpha.\ e$. Since $T' \equiv T$, then $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ (Lemma A.35).

(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. $\square$

A.37 Theorem [Progress (Theorem 5.16)]: If $\emptyset \vdash e : T$, then either

(1) $e \longrightarrow e'$, or
(2) e is a result r, i.e., a value or blame.

PROOF. By induction on the typing derivation.
(T_VAR): Contradictory: there is no derivation $\emptyset \vdash x : T$.
(T_CONST): $\emptyset \vdash k : \mathsf{ty}(k)$. In this case, $e = k$ is a result.
(T_OP): $\emptyset \vdash \mathrm{op}\,(e_1, \ldots, e_n) : \sigma(T)$, where $\mathsf{ty}(\mathrm{op}) = x_1 : T_1 \rightarrow \ldots \rightarrow x_n : T_n \rightarrow T$. 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. By value inversion (Lemma A.16), we know that we can type each of these values at the exact refinement types we need by T_EXACT. We assume that if $\mathrm{op}\,(v_1, \ldots, v_n)$ is well defined on values satisfying the refinements in its type, so E_OP applies.
(T_ABS): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \rightarrow T_2)$. In this case, $e = \lambda x{:}T_1.\ e_{12}$ is a result.
(T_APP): $\emptyset \vdash e_1\ e_2 : [e_2/x]\,T_2$; by inversion, $\emptyset \vdash e_1 : (x{:}T_1 \rightarrow T_2)$ 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 A.36) on $\emptyset \vdash e_1 : (x{:}T_1 \rightarrow T_2)$, there are two cases:

($e_1 = \lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$): In this case, $(\lambda x{:}T_1'.\ e_{12})\ v_2 \longrightarrow [v_2/x]e_{12}$ by E_BETA.
($e_1 = \langle T_1' \Rightarrow T_2' \rangle^l_\sigma$ and $\sigma(T_1') \equiv T_1$ and $\sigma(T_2') \equiv T_2$): We know that $T_1' \parallel T_2'$ by cast inversion (Lemma A.33). 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, $\langle B \Rightarrow B \rangle^l_\sigma\ v_2 \longrightarrow v_2$.
($T_2' = \alpha$ or $x{:}T_{21} \rightarrow T_{22}$ or $\forall \alpha.\ T_{22}$): Incompatible; contradictory.
($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = B$, then by E_CHECK, $\langle B \Rightarrow \{x{:}B \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle \sigma(\{x{:}B \mid e\}), \sigma([v_2/x]e), v_2 \rangle^l$. Otherwise, by E_PRECHECK, we have:
$$\langle B \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_{\sigma_1}\ (\langle B \Rightarrow T_2'' \rangle^l_{\sigma_2}\ v_2)$$
where $\sigma_1 = \sigma \mid \mathrm{AFV}(\{x{:}T_2'' \mid e\})$ and $\sigma_2 = \sigma \mid \mathrm{AFV}(T_2'')$.
($T_1' = \alpha$):
($T_2' = \alpha'$): It must be the case that $\alpha = \alpha'$, since $\alpha \parallel \alpha'$. By E_REFL, $\langle \alpha \Rightarrow \alpha \rangle^l_\sigma\ v_2 \longrightarrow v_2$.
($T_2' = B$ or $x{:}T_{21} \rightarrow T_{22}$ or $\forall \alpha.\ T_{22}$): Incompatible; contradictory.
($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = \alpha$, then by E_CHECK, $\langle \alpha \Rightarrow \{x{:}\alpha \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle \sigma(\{x{:}\alpha \mid e\}), \sigma([v_2/x]e), v_2 \rangle^l$. Otherwise,
$$\langle \alpha \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_{\sigma_1}\ (\langle \alpha \Rightarrow T_2'' \rangle^l_{\sigma_2}\ v_2)$$
where $\sigma_1 = \sigma \mid \mathrm{AFV}(\{x{:}T_2'' \mid e\})$ and $\sigma_2 = \sigma \mid \mathrm{AFV}(T_2'')$, by E_PRECHECK.

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

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

($T'_2 = x{:}T_{21} \to T_{22}$): If $T'_1 = T'_2$, then $\langle T'_1 \Rightarrow T'_1\rangle^l_\sigma\, v_2 \longrightarrow v_2$ by E_REFL. If not, then

$$\begin{array}{l}\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l_\sigma\, v_2 \longrightarrow \\ \lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ y : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x\ \mathsf{in}\ (\langle [y/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\,(v_2\, y))\end{array}$$

for some fresh variable y, where $\sigma_i = \sigma \mid \mathrm{AFV}(T_{1i}) \cup \mathrm{AFV}(T_{2i})$ ($i \in \{1,2\}$), by E_FUN.

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

$$\langle T'_1 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle T''_2 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l_{\sigma_1}\,(\langle T'_1 \Rightarrow T''_2\rangle^l_{\sigma_2}\, v_2)$$

, where $\sigma_1 = \sigma \mid \mathrm{AFV}(\{x{:}T''_2 \mid e\})$ and $\sigma_2 = \sigma \mid \mathrm{AFV}(T'_1) \cup \mathrm{AFV}(T''_2)$, by E_PRECHECK.

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

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

($T'_2 = \forall\alpha.\,T_{22}$): If $T'_1 = T'_2$, then $\langle T'_1 \Rightarrow T'_2\rangle^l_\sigma\, v_2 \longrightarrow v_2$ by E_REFL. If not, then $\langle \forall\alpha.\,T_{11} \Rightarrow \forall\alpha.\,T_{22}\rangle^l_\sigma\, v_2 \longrightarrow \Lambda\alpha.\ (\langle [\alpha/\alpha]\,T_{11} \Rightarrow T_{22}\rangle^l_\sigma\,(v_2\,\alpha))$ by E_FORALL.

($T'_2 = \{x{:}T''_2 \mid e\}$): If $T'_1 = T''_2$, then $\langle T'_1 \Rightarrow \{x{:}T'_1 \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T'_1 \mid e\}), \sigma([v_2/x]e), v_2\rangle^l$ by E_CHECK. If not, then $\langle T'_1 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle T''_2 \Rightarrow \{x{:}T''_2 \mid e\}\rangle^l_{\sigma_1}\,(\langle T'_1 \Rightarrow T''_2\rangle^l_{\sigma_2}\, v_2)$ where $\sigma_1 = \sigma \mid \mathrm{AFV}(\{x{:}T''_2 \mid e\})$ and $\sigma_2 = \sigma \mid \mathrm{AFV}(T'_1) \cup \mathrm{AFV}(T''_2)$, by E_PRECHECK.

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

($T'_2 = B$ or α or $x{:}T_{21} \to T_{22}$ or $\forall\alpha.\,T_{22}$): We see

$$\langle \{x{:}T''_1 \mid e'_1\} \Rightarrow T'_2\rangle^l_\sigma\, v_2 \longrightarrow \langle T''_1 \Rightarrow T'_2\rangle^l_{\sigma'}\, v_2$$

where $\sigma' = \sigma \mid \mathrm{AFV}(T''_1) \cup \mathrm{AFV}(T'_2)$, by E_FORGET.

($T'_2 = \{x{:}T''_2 \mid e'_2\}$): If $T'_1 = T'_2$, then we immediately have $\langle T'_1 \Rightarrow T'_2\rangle^l_\sigma\, v_2 \longrightarrow v_2$ by E_REFL. If $T'_1 = T''_2$, then

$$\langle T'_1 \Rightarrow \{x{:}T'_1 \mid e'_2\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T'_1 \mid e'_2\}), \sigma([v_2/x]e'_2), v_2\rangle^l$$

by E_CHECK. Otherwise,

$$\langle \{x{:}T''_1 \mid e'_1\} \Rightarrow \{x{:}T''_2 \mid e'_2\}\rangle^l_\sigma\, v_2 \longrightarrow \langle T''_1 \Rightarrow \{x{:}T''_2 \mid e'_2\}\rangle^l_{\sigma'}\, v_2$$

where $\sigma' = \sigma \mid \mathrm{AFV}(T''_1) \cup \mathrm{AFV}(\{x{:}T''_2 \mid e'_2\})$, by E_FORGET.

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

(T_TAPP): $\emptyset \vdash e_1\ T_2 : [T_2/\alpha]\,T_1$; 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 A.36). We can see $(\Lambda\alpha.\ e'_1)\ T_2 \longrightarrow [T_2/\alpha]e'_1$ by E_TBETA.

(T_CAST): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_\sigma : T_1 \to T_2$. In this case, $\langle T_1 \Rightarrow T_2\rangle^l_\sigma$ 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.

(T_CONV): $\emptyset \vdash e : T'$; by inversion, $\emptyset \vdash e : T$. By the IH, we see that $e \longrightarrow e'$ or $e = r$.
(T_EXACT): $\emptyset \vdash v : \{x{:}T \mid e\}$. Here, v is a result by assumption.
(T_FORGET): $\emptyset \vdash v : T$. Again, v is a result by assumption.
□

A.38 Lemma [Context and type well formedness (Lemma 5.17)]: (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. □

A.39 Theorem [Preservation (Theorem 5.18)]: 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): $\emptyset \vdash k : \mathsf{ty}(k)$. Contradictory—values don't step.
(T_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 A.38), $\emptyset \vdash \sigma(T)$. 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 $\sigma(T) \equiv \sigma'(T)$ (Lemma A.22).

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

(E_REDUCE/E_BETA): $(\lambda x{:}T_1.\ e_{12})\ v_2 \longrightarrow [v_2/x]e_{12}$. First, we have $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1' \to T_2')$. By inversion for lambdas (Lemma A.32), $x{:}T_1 \vdash e_{12} : T_2$. Moreover, $x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'$, which means $T_2 \equiv T_2'$ (Lemma A.19).
By substitution, $\emptyset \vdash [v_2/x]e_{12} : [v_2/x]T_2$. We then see that $[v_2/x]T_2 \equiv [v_2/x]T_2'$ (Lemma A.22), so T_CONV completes this case.
(E_REDUCE/E_REFL): $\langle T \Rightarrow T\rangle^l_\sigma\ v_2 \longrightarrow v_2$. By cast inversion (Lemma A.33), $\_{:}\sigma(T) \to \sigma(T) \equiv x{:}T_1' \to T_2'$ and $\emptyset \vdash \sigma(T)$. In particular, we have $\sigma(T) \equiv T_2'$ and $\sigma(T) \equiv T_1'$ (Lemma A.19). By substitutivity of conversion (Lemma A.22), $[v_2/x]\sigma(T) \equiv [v_2/x]T_2'$. Since $\sigma(T)$ is closed, we really know that $\sigma(T) \equiv [v_2/x]T_2'$.
By C_SYM and C_TRANS, we have $T_1' \equiv \sigma(T) \equiv [v_2/x]T_2'$. By T_CONV on $\emptyset \vdash v_2 : T_1'$, we have $\emptyset \vdash v_2 : [v_2/x]T_2'$.
(E_REDUCE/E_FORGET): $\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l_\sigma\ v_2 \longrightarrow \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}\ v_2$ where $\sigma' = \sigma \mid \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)$. We have $\sigma(T_1) = \sigma'(T_1)$ and $\sigma(T_2) = \sigma'(T_2)$. We restate the typing judgment and its inversion:

$$\begin{array}{l} \emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l_\sigma\ v_2 : [v_2/y]T_2' \\ \emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l_\sigma : (y{:}T_1' \to T_2') \\ \emptyset \vdash v_2 : T_1' \end{array}$$

By cast inversion (Lemma A.33), we know that $\emptyset \vdash \sigma(T_1)$ from $\emptyset \vdash \sigma(\{x{:}T_1 \mid e\})$ and $\emptyset \vdash \sigma(T_2)$—as well as $\_{:}\sigma(\{x{:}T_1 \mid e\}) \to \sigma(T_2) \equiv y{:}T_1' \to T_2'$ and $\{x{:}T_1 \mid e\} \parallel T_2$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$. Inverting this conversion (Lemma A.19), finding $\sigma(\{x{:}T_1 \mid e\}) \equiv T_1'$ and $\sigma(T_2) \equiv T_2'$. Then by T_CONV and C_SYM, $\emptyset \vdash v_2 : \sigma(\{x{:}T_1 \mid e\})$; by T_FORGET, $\emptyset \vdash v_2 : \sigma(T_1)$.
By T_CAST, we have $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'} : y{:}\sigma(T_1) \to \sigma(T_2)$, with $T_1 \parallel T_2$ iff $\{x{:}T_1 \mid e\} \parallel T_2$, and $\mathrm{AFV}(\sigma') \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$. (Note, however, that y does not appear in $\sigma(T_2)$—we write it to clarify the substitutions below.)

By T_APP, we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}\, v_2 : [v_2/y]\sigma(T_2)$. Since $\sigma(T_2) \equiv T_2'$, we have $[v_2/y]\sigma(T_2) \equiv [v_2/y]T_2'$ by Lemma A.22. We are done by T_CONV.

<u>(E_REDUCE/E_PRECHECK)</u>:

$$\begin{array}{l}\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \\ \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\,(\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2)\end{array}$$

where $\sigma_1 = \sigma \mid \mathrm{AFV}(\{x{:}T_2 \mid e\})$ and $\sigma_2 = \sigma \mid \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)$. We have $\sigma(T_1) = \sigma_2(T_1)$ and $\sigma(T_2) = \sigma_1(T_2) = \sigma_2(T_2)$ and $\sigma(\{x{:}T_2 \mid e\}) = \sigma_1(\{x{:}T_2 \mid e\})$. We restate the typing judgment and its inversion:

$$\begin{array}{l}\emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma\, v_2 : [v_2/y]T_2' \\ \emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma : y{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma A.33), $\emptyset \vdash \sigma(T_1)$ and $\emptyset \vdash \sigma(\{x{:}T_2 \mid e\})$, and $y{:}\sigma(T_1) \to \sigma(\{x{:}T_2 \mid e\}) \equiv y{:}T_1' \to T_2'$ Also, $T_1 \parallel \{x{:}T_2 \mid e\}$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$.

By inversion on $\emptyset \vdash \sigma(\{x{:}T_2 \mid e\})$, we find $\emptyset \vdash \sigma(T_2)$. Next, $T_1 \parallel T_2$ iff $T_1 \parallel \{x{:}T_2 \mid e\}$, and $\mathrm{AFV}(\sigma_2) \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$. Now by T_CAST, we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2} : y{:}\sigma(T_1) \to \sigma(T_2)$. Note, however, that y doesn't occur in $\sigma(T_2)$.

We can take the convertible function types and see that their parts are convertible: $\sigma(T_1) \equiv T_1'$ and $\sigma(\{x{:}T_2 \mid e\}) \equiv T_2'$. Using the first conversion, we find $\emptyset \vdash v_2 : \sigma(T_1)$ by T_CONV. By T_APP, $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2 : [v_2/y]\sigma(T_2)$, where $[v_2/y]\sigma(T_2) = \sigma(T_2)$.

By reflexivity of compatibility (easily proved) and SIM_REFINER, $\sigma(T_2) \parallel \sigma(\{x{:}T_2 \mid e\})$. We have well formedness derivations for both types and $\mathrm{AFV}(\sigma_1) \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$, as well, so $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1} : y{:}\sigma(T_2) \to \sigma(\{x{:}T_2 \mid e\})$ by T_CAST. Again, y does not appear in $\sigma(e)$ or $\sigma(T_2)$. By T_APP, we have $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\,(\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2) : [\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2/y]\sigma(\{x{:}T_2 \mid e\})$.

Since y isn't in $\sigma(\{x{:}T_2 \mid e\})$, we can see:

$$[\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2/y]\sigma(\{x{:}T_2 \mid e\}) = \sigma(\{x{:}T_2 \mid e\}) = [v_2/y]\sigma(\{x{:}T_2 \mid e\})$$

By substitutivity of conversion (Lemma A.22), we have $[v_2/y]\sigma(\{x{:}T_2 \mid e\}) \equiv [v_2/y]T_2'$. We can now apply T_CONV to find $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\,(\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2) : [v_2/y]T_2'$.

<u>(E_REDUCE/E_CHECK)</u>: $\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T \mid e\}), \sigma([v_2/x]e), v_2\rangle^{l'}$. Without loss of generality, we can suppose that x is fresh for σ. We restate the typing judgment with its inversion:

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

By cast inversion (Lemma A.33), $\emptyset \vdash \sigma(\{x{:}T \mid e\})$ and $\emptyset \vdash \sigma(T)$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$. Moreover, $y{:}\sigma(T) \to \sigma(\{x{:}T \mid e\}) \equiv y{:}T_1' \to T_2'$, where y doesn't occur in $\sigma(\{x{:}T \mid e\})$. This means that $\sigma(T) \equiv T_1'$ and $\sigma(\{x{:}T \mid e\}) \equiv T_2'$.

Using T_CONV and C_SYM with the first conversion shows $\emptyset \vdash v_2 : \sigma(T)$. By inversion on $\emptyset \vdash \sigma(\{x{:}T \mid e\})$, we see $x{:}\sigma(T) \vdash \sigma(e) : \mathsf{Bool}$. By term substitution (Lemma A.28), we find $\emptyset \vdash [v_2/x]\sigma(e) : \mathsf{Bool}$. Since $[v_2/x]\sigma = \sigma$, By Lemma A.4 (1), $[v_2/x]\sigma(e) = \sigma([v_2/x]e)$. Finally, $\sigma([v_2/x]e) \longrightarrow^* \sigma([v_2/x]e)$ by reflexivity (Lemma A.17).

T_CHECK (with WF_EMPTY) shows $\emptyset \vdash \langle \sigma(\{x{:}T \mid e\}), \sigma([v_2/x]e), v_2\rangle^l : \sigma(\{x{:}T \mid e\})$. By substitutivity of conversion (Lemma A.22), $[v_2/y]\sigma(\{x{:}T \mid e\}) \equiv [v_2/y]T_2'$. Since y doesn't occur in $\sigma(\{x{:}T \mid e\})$, we know that $[v_2/y]\sigma(\{x{:}T \mid e\}) = \sigma(\{x{:}T \mid$

$e\})$, so we can show that $\sigma(\{x{:}T \mid e\}) \equiv [v_2/y]\,T_2'$ by C_SYM, and now $\emptyset \vdash \langle \sigma(\{x{:}T \mid e\}), \sigma([v_2/x]e), v_2\rangle^l : [v_2/y]\,T_2'$ by T_CONV.

<u>(E_REDUCE/E_FUN)</u>:

$$\begin{array}{l}\langle x{:}T_{11} \rightarrow T_{12} \Rightarrow x{:}T_{21} \rightarrow T_{22}\rangle^l_\sigma\, v_2 \longrightarrow \\ \lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x\ \mathsf{in}\ (\langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\,(v_2\, z))\end{array}$$

for some fresh variable z, where $\sigma_i = \sigma \mid \mathrm{AFV}(T_{1i}) \cup \mathrm{AFV}(T_{2i})$ ($i \in \{1,2\}$). Without loss of generality, we can suppose that x is fresh for σ. We have $\sigma(T_{ji}) = \sigma_i(T_{ji})$ ($j \in \{1,2\}$). We restate the typing judgment with its inversion:

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

By cast inversion on the first derivation:

$$\begin{array}{ll}\emptyset \vdash \sigma(x{:}T_{11} \rightarrow T_{12}) & \emptyset \vdash \sigma(x{:}T_{21} \rightarrow T_{22}) \\ x{:}T_{11} \rightarrow T_{12} \parallel x{:}T_{21} \rightarrow T_{22} & \mathrm{AFV}(\sigma) \subseteq \emptyset \\ \multicolumn{2}{l}{\_{:}\sigma(x{:}T_{11} \rightarrow T_{12}) \rightarrow \sigma(x{:}T_{21} \rightarrow T_{22}) \equiv y{:}T_1' \rightarrow T_2'}\end{array}$$

By inversion of this last (Lemma A.19):

$$\sigma(x{:}T_{11} \rightarrow T_{12}) \equiv T_1' \qquad \sigma(x{:}T_{21} \rightarrow T_{22}) \equiv T_2'$$

So by T_CONV and C_SYM, we have $\emptyset \vdash v_2 : \sigma(x{:}T_{11} \rightarrow T_{12})$. By weakening (Lemma A.24), $x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash v_2 : \sigma(x{:}T_{11} \rightarrow T_{12})$.

By inversion of the well formedness of the function types:

$$\emptyset \vdash \sigma(T_{11}) \qquad x{:}\sigma(T_{11}) \vdash \sigma(T_{12}) \qquad \emptyset \vdash \sigma(T_{21}) \qquad x{:}\sigma(T_{21}) \vdash \sigma(T_{22})$$

By weakening (Lemma A.24), we find $x{:}\sigma(T_{21}) \vdash \sigma(T_{11})$ and $x{:}\sigma(T_{21}) \vdash \sigma(T_{21})$. By compatibility:

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

Since $\mathrm{AFV}(\sigma_1) \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$, we have $x{:}\sigma(T_{21}) \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1} : (\_{:}\sigma(T_{21}) \rightarrow \sigma(T_{11}))$ by T_CAST (notice that compatibility is symmetric, per Lemma A.26). By T_APP and T_VAR, we can see $x{:}\sigma(T_{21}) \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x : [x/\_]\sigma(T_{11}) = \sigma(T_{11})$. Again by T_APP, we have $x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash v_2\, z : [z/x]\sigma(T_{12})$. By weakening (Lemma A.24) and substitution (Lemma A.28), we have the following two derivations:

$$\begin{array}{l}x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash [z/x]\sigma(T_{12}) = [z/x]\sigma_2(T_{12}) = \sigma_2([z/x]\,T_{12}) \\ x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash \sigma(T_{22})\end{array}$$

By T_CAST and Lemma A.27:

$$x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2} : (y{:}[z/x]\sigma(T_{12}) \rightarrow \sigma(T_{22}))$$

Noting that y is free here. By T_APP:

$$\begin{array}{rl}x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash & \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l\,(v_2\, z) \\ & :\ [v_2\, z/y]\,T_{22} (= T_{22})\end{array}$$

Finally, by T_ABS and T_APP:

$$\emptyset \vdash \lambda x{:}\sigma(T_{21}).\ \begin{array}{l}\mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x\ \mathsf{in} \\ \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\,(v_2\, z)\end{array} \ : x{:}\sigma(T_{21}) \rightarrow \sigma(T_{22})$$

since $[\langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x/z]\sigma(T_{22}) = \sigma(T_{22})$.

Since y isn't in $x{:}\sigma(T_{21}) \rightarrow \sigma(T_{22})$, we can see that $x{:}\sigma(T_{21}) \rightarrow \sigma(T_{22}) = [v_2/y](x{:}\sigma(T_{21}) \rightarrow \sigma(T_{22}))$. Using this fact with substitutivity of conversion

(Lemma A.22), we find $x{:}\sigma(T_{21}) \to \sigma(T_{22}) \equiv [v_2/y]\,T_2'$. So—finally—by T_CONV we have:

$\emptyset \vdash \lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^{l}_{\sigma_1}\, x\ \mathsf{in}\ \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^{l}_{\sigma_2}\,(v_2\, z) : [v_2/y]\,T_2'$

(E_REDUCE/E_FORALL): $\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^{l}_{\sigma}\, v_2 \longrightarrow (\Lambda\alpha.\ \langle [\alpha/\alpha]\,T_1 \Rightarrow T_2\rangle^{l}_{\sigma}\,(v\,\alpha))$ Without loss of generality, we can suppose that α is fresh for σ. We restate the typing and its inversion:

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

By cast inversion (Lemma A.33):

$$\begin{array}{ll}\emptyset \vdash \sigma(\forall\alpha.T_1) & \emptyset \vdash \sigma(\forall\alpha.T_2) \\ \forall\alpha.T_1 \parallel \forall\alpha.T_2 & \mathrm{AFV}(\sigma) \subseteq \emptyset \\ \_{:}\sigma(\forall\alpha.T_1) \to \sigma(\forall\alpha.T_2) \equiv x{:}T_1' \to T_2' & \end{array}$$

By inversion of this last $\sigma(\forall\alpha.T_1) \equiv T_1'$ and $\sigma(\forall\alpha.T_2) \equiv T_2'$ (Lemma A.19). By T_CONV and C_SYM, $\emptyset \vdash v_2 : \sigma(\forall\alpha.T_1) = \forall\alpha.\sigma(T_1)$. By type variable weakening (Lemma A.25), WF_TVAR, and T_TAPP, we have:

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

. Note that $\sigma([\alpha/\alpha]\,T_1)$ may be different from $\sigma(T_1)$. By inversion of the universal type's well formedness, compatibility, type weakening (Lemma A.25), type substitution (Lemma A.31) and Lemma A.30:

$$\alpha \vdash \sigma([\alpha/\alpha]\,T_1) \qquad \alpha \vdash \sigma(T_2) \qquad [\alpha/\alpha]\,T_1 \parallel T_2$$

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

We know that $\forall\alpha.\sigma(T_2) \equiv T_2'$, so by type variable substitutivity of conversion (Lemma A.23), $[v_2/x]\forall\alpha.\sigma(T_2) \equiv [v_2/x]\,T_2'$. Since x isn't in $\forall\alpha.\sigma(T_2)$, we know that $\forall\alpha.\sigma(T_2) \equiv [v_2/x]\,T_2'$ (by way of Lemma A.22). Now we can see by T_CONV that $\emptyset \vdash \Lambda\alpha.\ (\langle [\alpha/\alpha]\,T_1 \Rightarrow T_2\rangle^{l}_{\sigma}\,(v\,\alpha)) : [v_2/x]\,T_2'$.

(E_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 $[e_2/x]\,T_2 \equiv [e_2'/x]\,T_2$ by reflexivity (Lemma A.17) and substitutivity (Lemma A.22).

(E_BLAME): $E\,[\Uparrow l] \longrightarrow \Uparrow l$ $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma A.38), 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): $\emptyset \vdash e\ T : [T/\alpha]\,T'$. By cases on the step taken.

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

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

By type abstraction inversion (Lemma A.34): $\alpha \vdash e' : T''$ and $\forall\alpha.T'' \equiv \forall\alpha.T'$; by inversion of this last (Lemma A.21), $T'' \equiv T'$.

By type variable substitution (Lemma A.31), $\emptyset \vdash [T/\alpha]e' : [T/\alpha]\,T''$. By type substitutivity of conversion (Lemma A.23), $[T/\alpha]\,T'' \equiv [T/\alpha]\,T'$. T_CONV gives us $\emptyset \vdash [T/\alpha]e' : [T/\alpha]\,T'$ as desired.

(E_COMPAT): $E\,[e] \longrightarrow E\,[e']$, where $E = [\,]\ T$. By the IH and T_TAPP.

(E_BLAME): $E\,[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma A.38), we know that $\emptyset \vdash T$. So we see $\emptyset \vdash \Uparrow l : T$ by T_BLAME.

(T_CAST): $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(T_1) \to \sigma(T_2)$. 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.

> (E_REDUCE/E_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 $[v/x]e_1 \longrightarrow^* \mathsf{true}$. By WF_EMPTY and the assumption that $[v/x]e \longrightarrow^* \mathsf{true}$, we can find $\emptyset \vdash v : \{x{:}T \mid e\}$ by T_EXACT.
> (E_REDUCE/E_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, $\emptyset \vdash \Uparrow l : \{x{:}T \mid e\}$.
> (E_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 $[v/x]e_1 \longrightarrow^* e$ and $e \longrightarrow e'$, then $[v/x]e_1 \longrightarrow^* e'$. Therefore, $\emptyset \vdash \langle \{x{:}T \mid e_1\}, e', v\rangle^l : \{x{:}T \mid e_1\}$ by T_CHECK.
> (E_BLAME): $E[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E[\Uparrow l] : T$ by assumption. By type well formedness (Lemma A.38), 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 $\emptyset \vdash e' : T'$.
(T_EXACT): $\emptyset \vdash v : \{x{:}T \mid e\}$. This case is contradictory—values don't step.
(T_FORGET): $\emptyset \vdash v : T$. This case is contradictory—values don't step.
□

A.3. Parametricity

A.40 Lemma [Term compositionality (Lemma 6.1)]: 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 $r_1 \sim r_2 : [e/x]T; \theta; \delta$.

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 $r_2 = v_2$. Only the refinement case is interesting.
($T = \{y{:}T' \mid e'\}$): We show both directions simultaneously, where $x \neq y$, i.e., y is fresh. By the IH for T', we know that

$$v_1 \sim v_2 : T'; \theta; \delta[e_1, e_2/x] \text{ iff } v_1 \sim v_2 : [e/x]T'; \theta; \delta.$$

It remains to show that the values satisfy their refinements.
That is, we must show:

$$\theta_1(\delta_1([v_1/y][e_1/x]e')) \longrightarrow^* \mathsf{true} \text{ iff } \theta_1(\delta_1([v_1/y][e/x]e')) \longrightarrow^* \mathsf{true}$$

$$\theta_2(\delta_2([v_2/y][e_2/x]e')) \longrightarrow^* \mathsf{true} \text{ iff } \theta_2(\delta_2([v_2/y][e/x]e')) \longrightarrow^* \mathsf{true}$$

So let:

$$\begin{array}{lclcl} \sigma_1 & = & \theta_1\delta_1[\delta_1(e)/x, v_1/y] & \longrightarrow^* & \theta_1\delta_1[e_1/x, v_1/y] = \sigma_1' \\ \sigma_2 & = & \theta_2\delta_2[\delta_2(e)/x, v_2/y] & \longrightarrow^* & \theta_2\delta_2[e_2/x, v_2/y] = \sigma_2' \end{array}$$

We have $\sigma_1 \longrightarrow^* \sigma_1'$ by reflexivity except for $\delta_1(e) \longrightarrow^* e_1$, which we have by assumption; likewise, we have $\sigma_2 \longrightarrow^* \sigma_2'$. Then $\sigma_i(e')$ and $\sigma_i'(e')$ coterminate (Lemma 5.5), and we are done. □

A.41 Lemma [Term Weakening/Strengthening]: If $x \notin T$, then $r_1 \sim r_2 : T; \theta; \delta[e_1, e_2/x]$ iff $r_1 \sim r_2 : T; \theta; \delta$.

PROOF. Similar to Lemma A.40. □

A.42 Lemma [Type Weakening/Strengthening]: If $\alpha \notin T$, then $r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ iff $r_1 \sim r_2 : T; \theta; \delta$.

PROOF. Similar to Lemma A.40. $\square$

A.43 Lemma [Type compositionality (Lemma 6.2)]:
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ iff $r_1 \sim r_2 : [T'/\alpha]T; \theta; \delta$.

PROOF. By induction on the (simple) structure of T, proving both directions simultaneously. As for Lemma A.40, 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 $r_2 = v_2$. Here, the interesting case is for function types, where we must deal with some asymmetries in the definition of the logical relation. We also include the case for quantified types.
$\underline{(T = x{:}T_1 \to T_2)}$: 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 $v_1 \sim v_2 : [T'/\alpha](x{:}T_1 \to T_2); \theta; \delta$. Let $v_1' \sim v_2' : [T'/\alpha]T_1; \theta; \delta$. We must show that $v_1\, v_1' \simeq v_2\, v_2' : [T'/\alpha]T_2; \theta; \delta[v_1', v_2'/x]$. 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 A.41 gives $R_{T',\theta,\delta} = R_{T',\theta,\delta[v_1',v_2'/x]}$ and so

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

By the IH on T_2, $r_1' \sim r_2' : [T'/\alpha]T_2; \theta; \delta[v_1', v_2'/x]$. By expansion, $v_1\, v_1'' \simeq v_2\, v_2'' : [T'/\alpha]T_2; \theta; \delta[v_1', v_2'/x]$.
$\underline{(\Leftarrow)}$: This case is similar: Given $v_1 \sim v_2 : [T'/\alpha](x{:}T_1 \to T_2); \theta; \delta$, 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, $v_1' \sim v_2' : [T'/\alpha]T_1; \theta; \delta$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : [T'/\alpha]T_2; \theta; \delta[v_1', v_2'/x]$. These normalize to $r_1' \simeq r_2' : [T'/\alpha]T_2; \theta; \delta[v_1', v_2'/x]$. By the IH on T_2,

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

Since $x \notin T'$, Lemma A.41 gives

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

Finally, by expansion

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

$\underline{(T = \forall\alpha'.T_0)}$: 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 $v_1 \sim v_2 : \forall\alpha'.([T'/\alpha]T_0); \theta; \delta$. 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, $r_1' \sim r_2'$:

$[T'/\alpha]T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By expansion, $v_1\, T_1 \simeq v_2\, T_2 : [T'/\alpha]T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. Then, $v_1 \sim v_2 : \forall \alpha'.([T'/\alpha]T_0); \theta; \delta$.
<u>($\Leftarrow$)</u>: This case is similar: given $v_1 \sim v_2 : \forall \alpha'.([T'/\alpha]T_0); \theta; \delta$, 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 $v_1\, T_1 \simeq v_2\, T_2 : [T'/\alpha]T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : [T'/\alpha]T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. 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, $v_1 \sim v_2 : \forall \alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$.

□

A.44 Lemma [Convertibility (Lemma 6.3)]: If $T_1 \equiv T_2$ then $r_1 \sim r_2 : T_1; \theta; \delta$ iff $r_1 \sim r_2 : T_2; \theta; \delta$.

PROOF. By induction on the conversion relation, leaving θ and δ general. The case where $r_1 = r_2 = \Uparrow l$ is immediate, so we only need to consider the case where $r_1 = v_1$ and $r_2 = v_2$.
<u>(C_VAR)</u>: It must be that $T_1 = T_2 = \alpha$, so we are done immediately.
<u>(C_BASE)</u>: It must be that $T_1 = T_2 = B$, so we are done immediately.
<u>(C_REFINE)</u>: We have that $T_1 = \{x{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{x{:}T_2' \mid \sigma_2(e)\}$, where $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$.
By cotermination (Lemma 5.5):

$$[v_1/x](\theta_1(\delta_1(\sigma_1(e)))) \longrightarrow^* \mathsf{true} \text{ iff } [v_1/x](\theta_1(\delta_1(\sigma_2(e)))) \longrightarrow^* \mathsf{true}$$
$$[v_2/x](\theta_2(\delta_2(\sigma_1(e)))) \longrightarrow^* \mathsf{true} \text{ iff } [v_2/x](\theta_2(\delta_2(\sigma_2(e)))) \longrightarrow^* \mathsf{true}.$$

Note that $[v_i/x](\theta_i(\delta_i(\sigma_j(e)))) = \sigma_j([v_i/x](\theta_i(\delta_i(e))))$ for $i, j \in \{1, 2\}$ since all substitutions here are closing.
<u>(C_FUN)</u>: We have that $T_1 = x{:}T_{11} \to T_{12} \equiv x{:}T_{21} \to T_{22} = T_2$.
Let $v_1' \sim v_2' : T_{21}; \theta; \delta$ be given; we must show that $v_1\, v_1' \simeq v_2\, v_2' : T_{22}; \theta; \delta[v_1', v_2'/x]$.
By the IH, we know that $v_1' \sim v_2' : T_{11}; \theta; \delta$, so we know that $v_1\, v_1' \simeq v_2\, v_2' : T_{12}; \theta; \delta[v_1', v_2'/x]$. We are done by another application of the IH.
The other direction is similar.
<u>(C_FORALL)</u>: We have that $T_1 = \forall \alpha.T_1' \equiv \forall \alpha.T_2' = T_2$.
Let R, T, and T' be given. We must show that $v_1\, T \simeq v_2\, T' : T_2'; \theta[\alpha \mapsto R, T, T']; \delta$. We know that $v_1\, T \simeq v_2\, T' : T_1'; \theta[\alpha \mapsto R, T, T']; \delta$, so we are done by the IH.
The other direction is similar.
<u>(C_SYM)</u>: By the IH.
<u>(C_TRANS)</u>: By the IHs. □

A.45 Lemma [Cast substitution]: If $\vdash \Gamma, x{:}T_1$, $\Gamma, x{:}T_1 \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : T_2$, and $\Gamma, x{:}T_2 \vdash e_1 \simeq e_2 : T$ then $\Gamma_1, x{:}T_1, \Gamma_2 \vdash e_1[\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x/x] \simeq e_2[\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x/x] : T$.

PROOF. Let $\Gamma \vdash \theta; \delta$. We must show that

$$\theta_1(\delta_1([\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x/x]e_1)) \simeq \theta_2(\delta_2([\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x/x]e_2)) : T; \theta; \delta.$$

Now $\theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T_1; \theta; \delta$ by definition. By assumption, $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x)) \simeq \theta_2(\delta_2(\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x)) : T_2; \theta; \delta$. So let δ' be $\delta[\delta_1(\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x), \delta_2(\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x)/x]$. We have $\Gamma, x{:}T_2 \vdash \theta; \delta'$, so by assumption we have that $\theta_1(\delta_1'(e_1)) \simeq \theta_2(\delta_2'(e_2)) : T; \theta; \delta$, which is the same as $\theta_1(\delta_1([\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x/x]e_1)) \simeq \theta_2(\delta_2([\langle T_1 \Rightarrow T_2 \rangle^l_\sigma\, x/x]e_2)) : T; \theta; \delta$, and we are done. □

A.46 Lemma [Cast reflexivity (Lemma 6.4)]: If $\vdash \Gamma$ and $T_1 \parallel T_2$ and $\Gamma \vdash \sigma(T_1) \simeq \sigma(T_1) : *$ and $\Gamma \vdash \sigma(T_2) \simeq \sigma(T_2) : *$ and $\mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(\_ {:} T_1 \to T_2)$.

PROOF. By induction on $cc(\langle T_1 \Rightarrow T_2 \rangle^l)$. We omit the majority of this proof, but we leave in the case when $T_1 = T_2$ to highlight the need for the E_REFL reduction rule.
$\underline{(T_1 = T_2)}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\langle \theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1)) \rangle^l_\sigma \simeq \langle \theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1)) \rangle^l_\sigma : \sigma(T_1 \to T_1); \theta; \delta.$$

Let $v_1 \sim v_2 : \sigma(T_1); \theta; \delta$. We must show that

$$\begin{array}{l} \langle \theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1)) \rangle^l_\sigma \; v_1 \simeq \\ \langle \theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1)) \rangle^l_\sigma \; v_2 : \sigma(T_1); \theta; \delta[v_1, v_2/z] \end{array}$$

for fresh z. By E_REFL, these normalize to $v_1 \sim v_2 : \sigma(T_1); \theta; \delta[v_1, v_2/z]$. Lemma A.41 finishes the case.
□

A.47 Theorem [Parametricity (Theorem 6.5)]: (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.
<u>(T_VAR)</u>: 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.
<u>(T_CONST)</u>: By the assumption that constants are assigned correct types.
<u>(T_OP)</u>: By the assumption that operators are assigned correct types (and the IHs for the operator's arguments).
<u>(T_ABS)</u>: We have $e = \lambda x{:}T_1.\ e_{12}$ and $T = x{:}T_1 \to T_2$ 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

$$\begin{array}{rcl} (\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\; v_1 & \longrightarrow & [v_1/x]\theta_1(\delta_1(e_{12})) \\ (\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\; v_2 & \longrightarrow & [v_2/x]\theta_2(\delta_2(e_{12})), \end{array}$$

it suffices to show

$$[v_1/x]\theta_1(\delta_1(e_{12})) \simeq [v_2/x]\theta_2(\delta_2(e_{12})) : T_2; \theta; \delta[v_1, v_2/x].$$

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.
<u>(T_APP)</u>: We have $e = e_1\, e_2$ and $\Gamma \vdash e_1 : x{:}T_1 \to T_2$ and $\Gamma \vdash e_2 : T_1$ and $T = [e_2/x]\, T_2$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(e_1\, e_2)) \simeq \theta_2(\delta_2(e_1\, e_2)) : [e_2/x]\, T_2; \theta; \delta.$$

By the IH,

$$\begin{array}{l} \theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : x{:}T_1 \to T_2; \theta; \delta, \text{ and} \\ \theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : T_1; \theta; \delta. \end{array}$$

These normalize to $r_{11} \sim r_{12} : x{:}T_1 \rightarrow T_2; \theta; \delta$ 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 we are done:

$$\begin{array}{rcl} \theta_1(\delta_1(e_1\ e_2)) & \longrightarrow^* & \Uparrow l \\ \theta_2(\delta_2(e_1\ e_2)) & \longrightarrow^* & \Uparrow l. \end{array}$$

So let $r_{ij} = v_{ij}$. 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 A.40,

$$r'_1 \sim r'_2 : [e_2/x]\,T_2; \theta; \delta.$$

By expansion, we can then see

$$\theta_1(\delta_1(e_1\ e_2)) \simeq \theta_2(\delta_2(e_1\ e_2)) : [e_2/x]\,T_2; \theta; \delta.$$

<u>(T_TABS)</u>: We have $e = \Lambda\alpha.\ e_0$ and $T = \forall\alpha. T_0$ 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

$$\begin{array}{rcl} \theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 & \longrightarrow & [T_1/\alpha]\theta_1(\delta_1(e_0)) \\ \theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 & \longrightarrow & [T_2/\alpha]\theta_2(\delta_2(e_0)) \end{array}$$

it suffices to show that

$$[T_1/\alpha]\theta_1(\delta_1(e_0)) \simeq [T_2/\alpha]\theta_2(\delta_2(e_0)) : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta.$$

Since $\Gamma, \alpha \vdash \theta[\alpha \mapsto R, T_1, T_2]; \delta$, the IH finishes the case with $\Gamma, \alpha \vdash e_0 \simeq e_0 : T_0$.
<u>(T_TAPP)</u>: We have $e = e_1\ T_2$ and $\Gamma \vdash e_1 : \forall\alpha. T_0$ and $\Gamma \vdash T_2$ and $T = [T_2/\alpha]\,T_0$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : [T_2/\alpha]\,T_0; \theta; \delta.$$

By the IH,

$$\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_1)) : \forall\alpha. T_0; \theta; \delta.$$

These normalize to $r_1 \sim r_2 : \forall\alpha. T_0; \theta; \delta$. If both results are blame, $\theta_1(\delta_1(e_1\ T_2))$ and $\theta_2(\delta_2(e_1\ T_2))$ also normalize to blame, and we are done. So 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 A.43, $r'_1 \sim r'_2 : [T_2/\alpha]\,T_0; \theta; \delta$. By expansion,

$$\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : [T_2/\alpha]\,T_0; \theta; \delta.$$

(T_Cast): We have $e = \langle T_1 \Rightarrow T_2 \rangle^l_\sigma$ and $\vdash \Gamma$ and $T_1 \parallel T_2$ and $\Gamma \vdash T_1$, $\Gamma \vdash T_2$ 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 A.46,

$$\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(T_1 \to T_2),$$

which is exactly what we were looking for.
(T_Blame): Immediate.
(T_Check): We have $e = \langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l$ and $\emptyset \vdash v : T_1$ and $\emptyset \vdash e_2 : \mathsf{Bool}$, $\vdash \Gamma$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $[v/x]e_1 \longrightarrow^* e_2$ 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, for some l'',

$$\begin{aligned} \theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) &\longrightarrow^* \Uparrow l'' \\ \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) &\longrightarrow^* \Uparrow l''. \end{aligned}$$

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,

$$[v/x]\theta_1(\delta_1(e_1)) \simeq [v/x]\theta_2(\delta_2(e_1)) : \mathsf{Bool}; \theta; \delta[v, v/x].$$

Then, we have

$$\begin{aligned} [v/x]\theta_1(\delta_1(e_1)) = [v/x]e_1 &\longrightarrow^* \mathsf{true} \\ [v/x]\theta_2(\delta_2(e_1)) = [v/x]e_1 &\longrightarrow^* \mathsf{true}. \end{aligned}$$

By definition, $v \simeq v : \{x{:}T_1 \mid e_1\}; \theta; \delta$. By expansion,

$$\theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) \simeq \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) : \{x{:}T_1 \mid e_1\}; \theta; \delta.$$

(T_Conv): By Lemma A.44.
(T_Exact): We have $e = v$ and $\emptyset \vdash v : T$ and $\emptyset \vdash \{x{:}T_0 \mid e_0\}$ and $[v/x]e_0 \longrightarrow^* \mathsf{true}$ 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

$$\begin{aligned} [v/x]\theta_1(\delta_1(e_0)) = [v/x]e_0 &\longrightarrow^* \mathsf{true} \\ [v/x]\theta_2(\delta_2(e_0)) = [v/x]e_0 &\longrightarrow^* \mathsf{true}. \end{aligned}$$

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.
□