A

Polymorphic Manifest Contracts, Revised and Resolved

TARO SEKIYAMA and ATSUSHI IGARASHI, Kyoto University
MICHAEL GREENBERG, Pomona College

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 $\{x{:}\alpha\ \mathsf{Stack} \mid \mathsf{not}\,(\mathsf{empty}\ x)\}$.

This article studies a polymorphic calculus with manifest contracts and establishes fundamental properties including type soundness and relational parametricity. Indeed, this is not the first work on polymorphic manifest contracts but existing calculi are not very satisfactory. Gronski et al. developed the SAGE language, which introduces polymorphism through the Type:Type discipline, but they do not study parametricity. Some authors of this paper have produced two separate works: Belo, Greenberg, Igarashi, and Pierce (ESOP 2011) and Greenberg (PhD thesis) studied polymorphic manifest contracts and parametricity, but their calculi have metatheoretical problems in the type conversion relations—indeed, they depend on a few conjectures, which turn out to be false. Our calculus is the first polymorphic manifest calculus with parametricity, depending on no conjectures—it resolves the issues in prior calculi with delayed substitution on casts.

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

CCS Concepts: •**Theory of computation** → **Lambda calculus; Pre- and post-conditions; Assertions; Operational semantics;** •**Software and its engineering** → **Functional languages; Polymorphism; Semantics; Syntax;**

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, corrections

1. INTRODUCTION

1.1. Motivation

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 nonempty list to a positive integer, where both "nonempty" and "positive" are expressed as code. 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

This article is based on Belo et al. [2011] and Chapter 3 of Greenberg [2013]. This work was supported in part by the JSPS Grant-in-Aid for Scientific Research (B) No. 25280024 from MEXT of Japan.
T. Sekiyama is currently affiliated with IBM Research - Tokyo.

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

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, Chapter 8]); in the manifest approach, contracts are types—the type system itself makes contracts 'manifest'—and dynamic contract checking is expressed by type coercions, which are more commonly called casts; static contract checking can be reduced to subtype checking.

Manifest contracts are a sensible choice for combining contracts and other type-based abstraction mechanisms, like 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 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 is not 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. Contracts on the ADT's interface do not allow `pred` to be applied to zero.

1.2. Polymorphic manifest contract calculus

A key device for studying type-based abstraction in functional programming is *parametric polymorphism*—for example, it is well known that polymorphism can encode ADTs [Mitchell and Plotkin 1985]. Gronski et al. studied manifest contracts in the presence of polymorphism by developing the SAGE language [Gronski et al. 2006], which supports manifest contracts and polymorphism, in addition to the type Dynamic [Abadi et al. 1989; Henglein 1992] and even the so-called "Type:Type" discipline [Cardelli 1986]. However, consequences of combining these features, in particular, interactions between manifest contracts and type abstraction (provided by parametric polymorphism), are not studied in depth in Gronski et al. [2006].

To study type abstraction for manifest contracts rigorously, Belo et al. [2011] developed a polymorphic manifest contract calculus $\mathrm{F_H}$, an extension of System F with manifest contracts, and investigated its properties, including type soundness and (syntactic) parametricity. For $\mathrm{F_H}$ to scale up to polymorphism, they made two technical contributions beyond earlier manifest calculi such as λ_{H} [Flanagan 2006], a simply typed manifest contract calculus. First, $\mathrm{F_H}$ gives the semantics of casts in the presence of so-called "general refinements," where the underlying type T in a refinement type $\{x{:}T \mid e\}$ can be an arbitrary type (not only base types like Bool and Int but also function, forall, and even refinement types), whereas earlier manifest calculi restrict refinements to base types. Support for general refinements is critical for polymorphism, because it means that we can refine an abstract datatype implemented by any type, not just base types. SAGE also allows arbitrary types to be refined but the semantics of casts relies on the type Dynamic, which is problematic for parametricity [Matthews and Ahmed 2008]. Second, Belo et al. have proposed a new, two-step, syntactic approach to formalizing manifest calculi. The first step is to establish fundamental properties such as type soundness for a calculus *without subsumption* (and subtyping), while earlier calculi [Knowles and Flanagan 2010; Greenberg et al. 2010] rest on subtyping and a denotational semantics of types in their construction. Technically, they replaced subtyping with a syntactic type conversion relation, which is required to show subject reduction in the presence of dependent function types. The lack of subsumption allows for an entirely syntactic metatheory but it also means omitting static contract checking. The second step is to give a static analysis to remove casts that never fail in order to compensate for the lack of static contract checking. In fact, Belo et al. prove *post facto* subtyping, examining a property called the Upcast Lemma, which says an upcast—a cast from one type to a supertype—is logically related (thus equivalent in a certain sense) to an identity function. The Upcast Lemma recovers a notion of static checking of contracts.

Unfortunately, however, the proofs of type soundness and parametricity of $\mathrm{F_H}$ turn out to be flawed and, worse, the properties themselves are later found to be false. In fact, the type conversion makes an inconsistent contract system; if a cast-free closed expression is well typed, then its type can be refined arbitrarily—e.g., integer 0 can be given type $\{x{:}\mathsf{Int} \mid x = 42\}$! These anomalies are first recognized as a false lemma about the type conversion relation. Greenberg [2013] fixed the false lemma by changing the conversion relation. Another key property of conversion, called *cotermination* and left as a conjecture in both Belo et al. [2011] and Greenberg [2013], also turns out to be wrong.[1] Inconsistency and the failure of type soundness and parametricity follow from counterexamples to these properties. As we will discuss in detail, the root cause of the problem can be attributed to the fact that substitution can badly affect how casts behave.

Table I. The status of properties of polymorphic manifest calculi.

	Belo et al. [2011] ($\mathrm{F_H}$)	Greenberg [2013]	This article ($\mathrm{F}_\mathrm{H}^\sigma$)
Lemma on conversion relation	✗	✓	✓
Cotermination	✗ (conjecture)	✗ (conjecture)	✓
Progress	✗	✗	✓
Preservation	?*	?*	✓
Parametricity	✗	✗	✓
Upcast Lemma	?*	?*	?

✓ $\cdots$ proved ?* $\cdots$ proved with flawed premises ✗ $\cdots$ flawed ? $\cdots$ unknown

1.3. Contributions

In this article, we introduce a new polymorphic manifest contract calculus $\mathrm{F}_\mathrm{H}^\sigma$ that resolves the technical flaws in $\mathrm{F_H}$. We call our calculus $\mathrm{F}_\mathrm{H}^\sigma$ because it takes the $\mathrm{F_H}$ from Belo et al. [2011] and Greenberg [2013] and introduces a new substitution semantics using *delayed substitutions*, which we write σ. Delayed substitutions are close to explicit substitutions [Abadi et al. 1991] but only substitutions on casts are explicit (and delayed) in $\mathrm{F}_\mathrm{H}^\sigma$. Although, in some work [Grossman et al. 2000; Ahmed et al. 2011], delayed substitutions, also called explicit bindings, have been used to represent syntactic "barriers" for type abstractions, we rather use them to determine how casts reduce statically. Thanks to delayed substitution, the semantics of $\mathrm{F}_\mathrm{H}^\sigma$ can choose cast reduction rules independently of substitution; this property is crucial when we prove cotermination. We can finally show that type soundness and parametricity all hold in $\mathrm{F}_\mathrm{H}^\sigma$—*without leaving any conjectures*. Consistency of the contract system of $\mathrm{F}_\mathrm{H}^\sigma$ is derived immediately from type soundness.

Table I summarizes the status of properties of polymorphic manifest calculi; the columns and rows represent properties and work on polymorphic manifest contracts, respectively. We wrote ✓ for properties that are proved, ?* for properties with proofs that are based on false premises, ✗ for properties that are flawed, and ? for properties we are unsure of. We have not investigated the Upcast Lemma in $\mathrm{F}_\mathrm{H}^\sigma$ because the first step of Belo et al.'s approach—namely, establishing fundamental properties for a manifest calculus without subsumption (hence static contract checking)—has turned out to be trickier than we initially thought and is worth independent treatment. While the proof of the Upcast Lemmas in Belo et al. [2011] and Greenberg [2013] would be themselves sensible,[2] they rest on unsound foundations. In this paper, we replace the foundations but stop short of replacing the Upcast Lemma. We believe (but have not proved) that we can prove it here, since the definition of parametricity is unchanged.

1.4. Outline of the article

This article is organized as follows. We start Section 2 with a brief history of manifest contract calculi (both monomorphic and polymorphic) and an (extended) example to motivate polymorphic manifest contracts and discuss their technical issues and our solutions. We define $\mathrm{F}_\mathrm{H}^\sigma$ in Section 3 and prove type soundness in Section 4, fixing Belo et al. [2011] with common-subexpression reduction from Greenberg [2013] and our novel use of delayed substitutions. We prove parametricity in Section 5; 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

[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 did not depend on cotermination, but this claim also turns out to be false.

[2]We discovered a small error in the proof itself of the Upcast Lemma in Greenberg's dissertation, but we have found a fix for $\mathrm{F}_\mathrm{H}^\sigma$.

contracts, resolving issues in prior versions of $\mathrm{F_H}$. Section 6 compares $\mathrm{F_H^\sigma}$ with two variants of polymorphic manifest contracts [Belo et al. 2011; Greenberg 2013] and presents counterexamples to broken properties in these earlier calculi. Finally, we discuss broader related work in Section 7, concluding in Section 8. The body of this paper states only key lemmas and theorems; the fairly detailed proofs are given in Appendix together with auxiliary lemmas and even more detailed proofs are in the online appendix.

2. OVERVIEW

This section first reviews manifest contract calculi [Flanagan 2006; Greenberg et al. 2010; Knowles and Flanagan 2010]—proposed as foundations of *hybrid type checking*, a synthesis of static and dynamic specification checking—and earlier polymorphic extensions [Belo et al. 2011; Greenberg 2013] with their technical challenges; then we describe problems in the earlier polymorphic calculi and our solutions.

2.1. Manifest contract calculus for hybrid type checking

Flanagan [2006] proposed hybrid type checking, a framework to combine static and dynamic verification techniques for modularly checking implementations against contract-based precise interface specifications, and formalized λ_H as a theoretical foundation to study hybrid type checking. Later work revised and refined those early ideas [Knowles and Flanagan 2010; Greenberg et al. 2010], naming the core dynamic checking framework a 'manifest contract calculus' (or simply, manifest calculus) [Greenberg et al. 2010].

Hybrid type checking reduces program verification to subtype checking problems, proving subtyping statically as much as possible and deferring checking to run time if a problem instance is not solved statically. We describe how these ideas are formalized in λ_H below; briefly, characteristic features of manifest contract calculi (in particular, early ones such as slightly different versions of λ_H) could be summarized as:

— *Type-based specifications:* refinement types (and dependent function types) to represent specifications;
— *Static checking:* subtyping to model static verification; and
— *Dynamic checking:* casts to model dynamic verification.

Type-based specifications. In λ_H, specifications are expressed in terms of types, more concretely, *refinement types* and *dependent function types*. A refinement type $\{x{:}B \mid e\}$ intuitively denotes the set of values v of base type B (e.g., Int, Bool, and so on) such that $[v/x]e$ reduces to true. In that type, e, also called a contract or a refinement, can be an *arbitrary* Boolean expression, so refinement types can represent any subset of the base-type constants as long as a constraint to specify the subset can be written as a program expression. For example, prime numbers can be represented as $\{x{:}\mathsf{Int} \mid \mathrm{prime}\ x\}$, using a primality test function prime, written in the same language as the program itself. A dependent function type $x{:}T_1 \to T_2$ denotes functions taking arguments v of domain type T_1 and returning values of codomain type $[v/x]T_2$. Dependent functions cleanly express the relation between inputs and outputs of a function. For example, $x{:}\mathsf{Int} \to \{y{:}\mathsf{Int} \mid y > x\}$ denotes functions that are strictly increasing, i.e., return an integer larger than the argument.

Manifest calculi need not have *arbitrary* Boolean expressions and dependent function types. For example, Ou et al. [2004] restrict predicates to be pure expressions and the blame calculus by Wadler and Findler [2009] supports only non-dependent function types. As we will discuss below, having arbitrary predicates and dependent functions significantly complicates metatheory. We will call a manifest calculus with both of these optional features a *full* manifest calculus.

Static checking. With these expressive types, program verification amounts to type checking, in particular, checking subtyping between refinement types. For example, to see if a prime number (of type $\{x{:}\mathsf{Int} \mid \mathrm{prime}\ x\}$) can be safely passed to a function expecting positive numbers (of type $\{x{:}\mathsf{Int} \mid x > 0\}$) is to see if the former type is a subtype of the latter. Informally, a refinement type $\{x{:}B \mid e_1\}$ is a subtype of $\{x{:}B \mid e_2\}$ when e_2 holds for any value of B satisfying e_1. Formally, supposing that we use σ to denote substitutions and write $\Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma$ to mean that σ is a closing substitution respecting $(\Gamma, x{:}\{x{:}B \mid \mathsf{true}\})$, Flanagan [2006] gives a subtyping rule for refinement types like:[3]

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

This formalization allows language designers to choose their favorite static checking methods because it states *what* static checking verifies, rather than how a specific static checking method works.

Dynamic checking. Unlike previous work on refinement types [Freeman and Pfenning 1991; Xi and Pfenning 1999; Mandelbaum et al. 2003; Ou et al. 2004], however, manifest contracts' predicate language is very expressive—in fact, too expressive to be decidable. Flanagan's approach to undecidable subtyping is to defer subtyping checks until runtime, inserting *casts* where subtyping cannot be decided, rather than rejecting a program. More concretely, if static checking cannot decide whether the type T_1 of a given expression e is a subtype of T_2, then the compiler inserts a *cast*—written $\langle T_1 \Rightarrow T_2\rangle^l$—from T_1 (the *source type*) to T_2 (the *target type*) and yields $\langle T_1 \Rightarrow T_2\rangle^l\ e$. At runtime, the cast's evaluation checks whether (the value of) e (of type T_1) can behave like a value of type T_2. The superscript l is called a *blame label*, an abstract source location used to differentiate between different casts and identify the source of failures.

We briefly explain how casts work in simple cases. At refinement types, casts either return the value they are applied to, or abort program execution by raising "blame" (a kind of uncatchable exception), indicating which supposed subtyping turned out to be false. For example, consider a cast from positive integers $\{x{:}\mathsf{Int} \mid x > 0\}$ to odd integers $\{x{:}\mathsf{Int} \mid \mathrm{odd}\ x\}$. If we apply cast $\langle\{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathrm{odd}\ x\}\rangle^l$ to 5, we expect to get 5 back, since 5 is an odd integer (that is, $\mathrm{odd}\ 5$ evaluates to true). So,

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

Then, 5 can be typed at $\{x{:}\mathsf{Int} \mid \mathrm{odd}\ x\}$. On the other hand, suppose we apply the same cast to 2. This cast fails, since 2 is even. When the cast fails, it will raise blame with its label:

$$\langle\{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathrm{odd}\ x\}\rangle^l\, 2 \longrightarrow^* \Uparrow l.$$

Casts between dependent function types are also made possible in λ_{H} by adapting higher-order contracts by Findler and Felleisen [2002], running domain checks contravariantly and codomain checks covariantly (with an asymmetric substitution; see Section 3).

Type soundness of λ_H. Proving syntactic type soundness of a full calculus (such as λ_{H}) via progress and preservation is tricky. We identify two main issues here.

The first issue is how to allow values to be typed at refinements they satisfy. For example, the type system should be able to give integer 2 type $\{x{:}\mathsf{Int} \mid \mathsf{true}\}$, $\{x{:}\mathsf{Int} \mid$

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

$\mathrm{even}\ x\}$, or $\{x{:}\mathsf{Int} \mid \mathrm{prime}\ x\}$. Subtyping is one way to assign 2 more than one type, by way of "selfified" types [Ou et al. 2004]. The selfified type for a constant is the *most specific type*—e.g., the selfified type of an integer n is $\{x{:}\mathsf{Int} \mid x = n\}$. For example, if $\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\ n \longrightarrow^* n$, then n can be given type $\{x{:}\mathsf{Int} \mid x > 0\}$ by using the subtyping rule above, because inhabitants of the selfified type of n are only n and the dynamic check has ensured that $n > 0$ holds.

The second issue is standard in a calculus with dependent function types: if e_1 evaluates to e_2, the type system must allow terms of type $[e_1/x]\,T$ to be typed at $[e_2/x]\,T$, too, and vice versa to show preservation. Let us consider the case for a function application $v_1\ e_2 \longrightarrow v_1\ e_2'$. Since v_1 is at a function position, its type takes the form $x{:}\,T_1 \to T_2$. The codomain type of a function is dependent on an argument to the function, so types of $v_1\ e_2$ and $v_1\ e_2'$ would be $[e_2/x]\,T_2$ and $[e_2'/x]\,T_2$, respectively. Since preservation says that evaluation preserves types of well typed terms, $v_1\ e_2'$ has to be typed also at $[e_2/x]\,T_2$.

A typical solution found in dependent type theory [de Bruijin 1980; Barendregt 1992; Harper et al. 1993] is to introduce a type equivalence relation, which is congruence closed under (β or sometimes $\beta\eta$) reduction. Ou et al. [2004] address this issue with subtyping; they show that, for any pure expressions e_1 and e_2, if $e_1 \longrightarrow e_2$, then $[e_2/x]\,T$ is a subtype of $[e_1/x]\,T$. It is not clear, however, how $[e_1/x]\,T$ and $[e_2/x]\,T$ should be related in a full manifest calculus mainly due to the above-mentioned subtyping rule for refinement types and the fact that computation is effectful (recall that blame is an uncatchable exception). Unfortunately, earlier work is not fully satisfactory in this regard. In fact, neither Flanagan [2006] nor Knowles and Flanagan [2010] discusses this issue and Greenberg et al. [2010] sidesteps it by showing only *semantic type soundness* using a logical predicate technique, which is motivated by an issue with monotonicity—see Section 2.3. (Knowles and Flanagan [2010] and Greenberg et al. [2010] prove, though, a closely related property that, if $e_1 \longrightarrow e_2$, then $[e_1/x]\,T$ and $[e_2/x]\,T$ are *semantic* subtypes of each other.)

In short, there is no fully satisfactory proof of syntactic type soundness of a full manifest calculus. Semantic type soundness is fine, but is hard to extend as features are added to the calculus. Thus, a more syntactic proof is desirable. Belo et al. [2011] tried proving type soundness in a more syntactic manner when they extend a manifest calculus to parametric polymorphism.

2.2. Motivating example of parametrically polymorphic manifest contracts

Parametric polymorphism is a cornerstone of reusability in functional programming. For example, polymorphism can encode existentials, which are crucial for defining abstract datatypes and expressing modularity. In our context, manifest contracts are also used to specify precise interfaces of modules by refining existentials.

To better understand benefits of combining polymorphism with manifest contracts, we take a close look at NAT, an abstract datatype of natural numbers, given in Section 1. By using existenital types, natural numbers are represented as:

$$\mathsf{NAT} : \exists\alpha.(\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{isZ} : (\alpha \to \mathsf{Bool})) \times (\mathsf{pred} : \alpha \to \alpha)$$

where we name components of (nested) pairs—these are dependent sum types $(x : T_1) \times T_2$, where x can appear in T_2 and refers to the value of the first component.[4] The constructors zero and succ are standard; the operator isZ determines whether a natural is zero; the operator pred yields the predecessor. We omit the implementation, a standard Church encoding, where $\alpha = \forall\beta.\beta \to (\beta \to \beta) \to \beta$.

As we saw in Section 1, the standard representation of the naturals is *inadequate* with respect to the mathematical natural numbers, in particular with respect to pred.

[4]Dependent sums usually do not name their last component, but we do here for convenience.

In math, $\mathsf{pred}\,\mathsf{zero}$ is undefined, but the implementation will return zero. The NAT's 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 $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{isZ}\,x)\}$. With contracts, this is easy enough: the interface NAT_I is given as

$$\exists\alpha.(\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{isZ} : (\alpha \to \mathsf{Bool})) \times (\mathsf{pred} : \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{isZ}\,x)\} \to \alpha).$$

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

$$\mathsf{unpack}\ \mathsf{NAT} : \mathsf{NAT}_I\ \mathsf{as}\ \alpha, n\ \mathsf{in}\ n.\mathsf{isZ}\,(n.\mathsf{pred}\,(n.\mathsf{zero})) : \mathsf{Bool}$$

We have "unpacked" the ADT to make its type available as α and used the dot notation to clarify constructors and operators specified. We then ask if the predecessor of 0 is 0, running $n.\mathsf{isZ}\,(n.\mathsf{pred}\,(n.\mathsf{zero}))$. The inner application is *not well typed*! We have that $\mathsf{zero} : \alpha$, but the domain type of pred is $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{isZ}\,x)\}$. To make the application well typed, we must insert a cast:

$$\begin{array}{l}\mathsf{unpack}\ \mathsf{NAT} : \mathsf{NAT}_I\ \mathsf{as}\ \alpha, n\ \mathsf{in} \\ \qquad n.\mathsf{isZ}\,(n.\mathsf{pred}\,(\langle \alpha \Rightarrow \{x{:}\alpha \mid \mathsf{not}\,(n.\mathsf{isZ}\,x)\}\rangle^l\, n.\mathsf{zero})) : \mathsf{Bool}\end{array}$$

Naturally, this cast will ultimately raise $\Uparrow l$, because $\mathsf{not}\,(n.\mathsf{isZ}\,n.\mathsf{zero}) \longrightarrow^* \mathsf{false}$. This way, abstract datatypes can impose constraints on their *use*—in this example, the use of pred.

Manifest contracts also can impose constraints on the *implementation* of the abstract type so that users of the abstract datatype can expect the implementation to return values satisfying the constraints. For example, consider a more accurate interface of pred: $\mathsf{pred}\,x$ will always be less than x. That is, when we extend the NAT's interface with a binary "less than" operator lt, the result $\mathsf{pred}\,x$ has the refined type $\{y{:}\alpha \mid \mathsf{lt}\,y\,x\}$. We can specify this fact with the interface:[5]

$$\exists\alpha.\ \ldots \times (\mathsf{lt} : \alpha \to \alpha \to \mathsf{Bool}) \times (\mathsf{pred} : x{:}\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{isZ}\,x)\} \to \{y{:}\alpha \mid \mathsf{lt}\,y\,x\})$$

The pred function's contract requires that pred's argument is nonzero and that pred returns a result less than the argument.

How can we write an implementation to meet this interface? By putting casts in the implementation. We can impose the contract on pred 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 in terms of the standard, unrefined implementation, pred':

$$\mathsf{pred}\ =\ \langle \mathsf{nat} \to \mathsf{nat} \Rightarrow x{:}\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{isZ}\,x)\} \to \{y{:}\mathsf{nat} \mid \mathsf{lt}\,y\,x\}\rangle^l\, \mathsf{pred}'$$

Note, however, that the cast on pred' will never actually check its domain contract at runtime: if we unfold the domain contract contravariantly, we see that $\langle\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{isZ}\,x)\} \Rightarrow \mathsf{nat}\rangle^l$ is a no-op, because we are casting out of a refinement. Instead, clients of NAT can only call pred with terms that are typed at $\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{isZ}\,x)\}$, i.e., by checking that values are nonzero with a cast into pred's input type. The codomain contract on pred, however, could fail if pred' mis-implemented predecessor.

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

[5]Precisely speaking, we need to apply a cast $\langle\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{isZ}\,x)\} \Rightarrow \alpha\rangle^l$ to x in the codomain contract of pred to make the contract well typed. We omit the cast because it will always succeed, returning x.

recalls Findler and Felleisen's seminal idea of client and server blame [Findler and Felleisen 2002].

Readers interested in other examples in polymorphic manifest contracts are referred to Greenberg's thesis [Greenberg 2013]. The thesis gives another, longer and more detailed example in polymorphic manifest contracts and shows that contracts with polymorphism can enforce sophisticated typing disciplines easily.

2.3. Key ideas in polymorphic manifest contract calculus F_H

The full manifest calculus F_H [Belo et al. 2011] combined the type abstraction of parametric polymorphism with manifest contracts. This section describes key ideas in that work, namely refinement types with arbitrary underlying types and subsumption-free formalization, and the next presents technical flaws in the metatheory of F_H.

Polymorphism and general refinements. Adding polymorphism to manifest contracts is not as simple as it might appear. The crux of the matter is this: we need to be able to write $\{x{:}\alpha \mid e\}$ for refinements to interact with abstract datatypes in a useful way. A question here is: What types can be instantiated for the type variable α? Earlier manifest calculi restrict refinements to base types, forbidding refinements of function types like $\{f{:}(\mathsf{Int} \to \mathsf{Int}) \mid f\,0 = 0\}$. However, this restriction is severe and limits the expressiveness of types excessively. For example, let us remember the NAT example. Since the implementation type is $\forall\beta.\beta \to (\beta \to \beta) \to \beta$, the predecessor function `pred` over naturals has to be implemented as a function of type

$$\{x{:}\forall\beta.\beta \to (\beta \to \beta) \to \beta \mid \mathsf{not}\,(\mathsf{isZ}\,x)\} \to (\forall\beta.\beta \to (\beta \to \beta) \to \beta),$$

in which, to restrict arguments to be nonzero, the domain type refines the Church natural number type $\forall\beta.\beta \to (\beta \to \beta) \to \beta$ by substituting it for the abstract type. Systems without general refinements would reject this type as *ill-formed*, because the underlying type is not a base type.

Thus, F_H supports *general refinements*, which allow type variables α to be instantiated with *any* type, that is, not only base types like Bool and Int but also function, forall, and even refinement types. Introducing general refinements calls for a new semantics for casts: how do casts evaluate? In our system, a cast $\langle T_1 \Rightarrow T_2\rangle^l$ evaluates in several steps (we describe it in detail in Section 3). Roughly speaking, the semantics forgets refinements in T_1 and then starts checking refinements in T_2 from the inside out. The cast semantics of F_H skips some refinement checks that appear to be unnecessary. For example, reflexive casts of the form $\langle T \Rightarrow T\rangle^l$ just disappear—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 seems that the cast disappears like the identity function.

We believe that F_H^σ is the first sound polymorphic manifest calculus with general refinement types. As we mentioned in the introduction, SAGE also allows any type to be refined; however, in SAGE, the source type in a cast is always Dynamic. While this makes the cast semantics much simpler, parametricity in the presence of Dynamic would not be straightforward (for a summary of the difficulty, see related work in Section 7). Belo et al. [2011] also has general refinements but it has some metatheoretical problems; Sekiyama et al. [2015] have not dealt with polymorphism.

Subsumption-free formulation. Although subtyping plays a cruicial role in manifest calculi, it comes with some metatheoretic baggage, as described by Knowles and Flanagan [2010] and Greenberg et al. [2010]. The issue is that rules of the type system in Flanagan [2006] are not monotonic—in particular, the subtyping rule for refinement types quantifies over all well formed closing substitutions, which in turn refer to well typedness—and so it is not clear that the type system is even well defined. Knowles and Flanagan [2010] and Greenberg et al. [2010] avoided the issue by giving denotational semantics (namely, logical predicates) of types and changing the problematic subtyping rule so that it refers to the denotations instead of well typedness. One (philosophical) problem is that soundness of the type system with respect to the denotational semantics has to be shown *before* soundness with respect to the operational semantics. Another, perhaps more serious problem is that the denotational approach is generally harder to scale than standard syntactic methods (i.e., progress and preservation), when we consider other features such as polymorphism. We discuss it in more detail in Section 7.2.

$\mathrm{F_H}$ addresses this issue by dropping subsumption (and hence subtyping) from the type system. Since subtyping is removed, it is easy to see that the type system is well defined. However, removing subtyping raises the two issues for type soundness again and, additionally, another issue about how to deal with static verification, which is based on subtyping in the original hybrid checking framework.

For the type soundness issues, Belo et al. introduce a special typing rule to give values any refinement they satisfy and a type conversion relation, which is based on (call-by-value) parallel reduction.[6] With the type conversion relation, $[e_1/x]\,T$ and $[e_2/x]\,T$ are convertible if $e_1 \longrightarrow e_2$ and a typing rule that allows terms to be retyped at convertible types is substituted for the subsumption rule. Using such a type system, they claim to have "proved" type soundness in an entirely syntactic manner—via progress and preservation—and also parametricity based on syntactic logical relations.

Although the resulting system can be formalized without resting on denotational semantics, the lack of subsumption means that all refinements in a well typed program will be checked at runtime. As we have already mentioned in Section 1, Belo et al. recover static verification by introducing subtyping *post facto* and examining sufficient conditions to eliminate casts.

2.4. Flaws in $\mathrm{F_H}$—and how we solve them

Unfortunately, as mentioned in Section 1, a few properties required to show type soundness and parametricity turn out to be false. We will discuss the flawed properties with their counterexamples in detail in Section 6 but, in essence, the source of anomaly is that substitutions, which affect how casts behave, badly interact with the type conversion. As we discussed above, for preservation, two types $[e_1/x]\,T$ and $[e_2/x]\,T$ should be convertible if $e_1 \longrightarrow e_2$. Naively allowing this, however, will cause two refinement types $\{x{:}T \mid e_1\}$ and $\{x{:}T \mid e_2\}$ to be convertible (via $\{x{:}T \mid \Uparrow l\}$) for *any* Boolean terms e_1 and e_2. So, $\mathrm{F_H}$'s (static) contract system is inconsistent in the sense that a well typed cast-free term can be given any refinement—e.g., 0 can be given $\{x{:}\mathsf{Int} \mid x = 42\}$—and, worse, the inconsistency implies the lack of type soundness in $\mathrm{F_H}$—that is, an expression of a refinement type $\{x{:}T \mid e\}$ may result in a value not satisfying the predicate e.

The following shows a convertibility derivation to relate $\{x{:}T \mid e\}$ and $\{x{:}T \mid \Uparrow l\}$ for any e (here, given (closed) expression e, we write Int_e for $\{z{:}\mathsf{Int} \mid e\}$ and && stands for

[6]Belo et al. [2011] do not really show a formal definition of type conversion; it appears in Greenberg [2013].

Boolean conjunction).

$$
\begin{tikzcd}
\{x{:}T \mid e\} \arrow[d, phantom, "\equiv"', "\substack{\text{reflexive cast}\\ 42 > 0 \mathbin{\&\&} e \longrightarrow^* e}"'] & & \{x{:}T \mid \Uparrow l\} \arrow[d, phantom, "\equiv", "(\langle \mathsf{Int}_{\mathsf{false}} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e \longrightarrow^* \Uparrow l"'] \\
\{x{:}T \mid (\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \arrow[d, equal] & & \{x{:}T \mid (\langle \mathsf{Int}_{\mathsf{false}} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \arrow[d, equal] \\
[5 = 0/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \arrow[rr, phantom, "\equiv", "5 = 0 \longrightarrow \mathsf{false}"'] & & [\mathsf{false}/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\}
\end{tikzcd}
$$

The crux of this example is that substitution of $5 = 0$ for y yields a reflexive cast, while that of false for y yields a failing cast. Actually, the two intermediate types are *ill-formed*, because 42 cannot be given type $\mathsf{Int}_{5=0}$ or $\mathsf{Int}_{\mathsf{false}}$—the source types of the casts. Nevertheless, we cannot exclude such nonsense terms *before* defining our typing relation, so we must examine properties of a type conversion relation in the untyped setting *until we prove type soundness*.

$\mathrm{F}_{\mathrm{H}}^{\sigma}$ corrects this anomaly; in $\mathrm{F}_{\mathrm{H}}^{\sigma}$,

$$\{x{:}T \mid (\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \not\equiv \{x{:}T \mid (\langle \mathsf{Int}_{\mathsf{false}} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\},$$

avoiding $\{x{:}T \mid e\} \equiv \{x{:}T \mid \Uparrow l\}$, whereas

$$[5 = 0/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \;\equiv\; [\mathsf{false}/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\}$$

does hold. At first, these (in)equations seem contradictory because the first type $\{x{:}T \mid (\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\}$ and the third $[5 = 0/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\}$ are usually syntactically equal and so are the second and fourth. In fact, $\mathrm{F}_{\mathrm{H}}^{\sigma}$ distinguishes both pairs syntactically and obtains desirable type conversion, as illustrated below.

$$
\begin{tikzcd}
\{x{:}T \mid e\} \arrow[d, phantom, "\equiv"] & \{x{:}T \mid \Uparrow l\} \arrow[d, phantom, "\equiv"] \\
\{x{:}T \mid (\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \arrow[d, phantom, "\not\equiv"] & \{x{:}T \mid (\langle \mathsf{Int}_{\mathsf{false}} \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \arrow[d, phantom, "\not\equiv"] \\
[5 = 0/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\} \arrow[r, phantom, "\equiv"] & [\mathsf{false}/y]\{x{:}T \mid (\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l\, 42) > 0 \mathbin{\&\&} e\}
\end{tikzcd}
$$

More specifically, casts $[5 = 0/y]\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l$ and $[\mathsf{false}/y]\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l$ are distinguished from $\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l$ and $\langle \mathsf{Int}_{\mathsf{false}} \Rightarrow \mathsf{Int}_{5=0} \rangle^l$, respectively. This is achieved by changing the syntax and semantics of casts so that substitution does not affect how casts behave.

To distinguish $[5 = 0/y]\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l$ and $\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l$, $\mathrm{F}_{\mathrm{H}}^{\sigma}$ uses *delayed substitutions* σ, which are also used to ensure that substitution does not interfere with how casts evaluate. First, cast expressions are augmented with delayed substitutions and take the form $\langle T_1 \Rightarrow T_2 \rangle^l_\sigma$. (We often omit σ when it is empty.) Second, a substitution applied to a cast is *not* forwarded to its target and source types immediately but is instead stored as delayed substitutions—this is the reason why σ is called "delayed." For example, when term $5 = 0$ is substituted for y in $\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l$, the result is $\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l_{\{y \mapsto 5=0\}}$ where $\{y \mapsto 5 = 0\}$ maps y to $5 = 0$. Delayed substitutions attached to casts are ignored when deciding what steps to take to check values. Thus, $\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l_{\{y \mapsto 5=0\}}$ does not disappear, even when $[5 = 0/y]\mathsf{Int}_y$

Types and contexts

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

Substitutions

$$\sigma \ \in \ (\mathsf{TmVar} \overset{\text{fin}}{\rightharpoonup} \mathsf{Tm}) \times (\mathsf{TyVar} \overset{\text{fin}}{\rightharpoonup} \mathsf{Ty})$$

Terms, values, and results

$$\begin{aligned}\mathsf{Tm} \ni e \ ::= \ & x \mid k \mid \mathrm{op}\,(e_1, \ldots, e_n) \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid e_1\, e_2 \mid e\ T \mid \\ & \langle T_1 \Rightarrow T_2 \rangle^l_\sigma \mid \Uparrow l \mid \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l\end{aligned}$$
$$v \ ::= \ k \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid \langle T_1 \Rightarrow T_2 \rangle^l_\sigma$$
$$r \ ::= \ v \mid \Uparrow l$$

Evaluation contexts

$$E \ ::= \ [\,]\, e_2 \mid v_1\, [\,] \mid [\,]\ T \mid \langle \{x{:}T \mid e\}, [\,], v \rangle^l \mid \mathrm{op}(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)$$

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

and $\mathsf{Int}_{5=0}$ are syntactically equal; instead, a check to see if $5 = 0$ evaluates to true will run and the cast will raise blame eventually. Thanks to delayed substitution, we can distinguish $[5 = 0/y]\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l$ and $\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l$ because $[5 = 0/y]\langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l = \langle \mathsf{Int}_y \Rightarrow \mathsf{Int}_{5=0} \rangle^l_{\{y \mapsto 5=0\}}$ is not syntactically equivalent to $\langle \mathsf{Int}_{5=0} \Rightarrow \mathsf{Int}_{5=0} \rangle^l$.

Delayed substitution makes the contract system of $\mathrm{F}_{\mathrm{H}}^{\sigma}$ consistent—e.g., if 0 is given $\{x{:}\mathsf{Int} \mid e\}$ in the system, then $[0/x]e$ always returns true. The consistency, formalized as a lemma called *value inversion* in Section 4, makes it possible to establish fundamental properties, including type soundness, of $\mathrm{F}_{\mathrm{H}}^{\sigma}$.

3. DEFINING $\mathrm{F}_{\mathrm{H}}^{\sigma}$

3.1. Syntax

$\mathrm{F}_{\mathrm{H}}^{\sigma}$'s syntax extends System F with features from manifest contracts (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 that the typing rules for constants and the typing and evaluation rules for operations respect this set, along with some formal requirements (Section 3.3). 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 $\mathrm{F}_{\mathrm{H}}^{\sigma}$ are more general than existing manifest calculi (except for SAGE [Gronski et al. 2006]) 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 ordering of effects. Although we have used existential types and dependent sums in examples, $\mathrm{F}_{\mathrm{H}}^{\sigma}$ does not have them as primitives, because they can be encoded by using type abstractions as usual.

$$\begin{array}{rcl}
\exists\alpha.\,T & = & \forall\alpha'.(\forall\alpha.\,T \to \alpha') \to \alpha' \\
\mathsf{pack}\ \langle T_1, e\rangle\ \mathsf{as}\ \exists\alpha.\,T_2 & = & \Lambda\alpha'.\,\lambda f{:}(\forall\alpha.\,T_2 \to \alpha').\,f\ T_1\ e \\
\mathsf{unpack}\ e_1 : \exists\alpha.\,T_1\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2 : T_2 & = & e_1\ T_2\,(\Lambda\alpha.\,\lambda x{:}T_1.\,e_2) \\
\\
(x : T_1) \times T_2 & = & \forall\alpha.(x{:}T_1 \to T_2 \to \alpha) \to \alpha \\
(e_1, e_2)_{(x:T_1)\times T_2} & = & \Lambda\alpha.\,\lambda f{:}(x{:}T_1 \to T_2 \to \alpha).\,f\ e_1\ e_2.
\end{array}$$

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

As we have already discussed in the last section, casts in $\mathrm{F}_{\mathrm{H}}^{\sigma}$ are of the form $\langle T_1 \Rightarrow T_2\rangle_{\sigma}^{l}$, where the delayed substitution σ is formally a pair of substitutions from term and type variables to terms and types, respectively. 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, which can implement active checks as ordinary conditionals, but we keep it around as a technical aid to our syntactic proof of preservation. The value in the third position can be any value, not just a constant according to generalization of refinement types. If checking the refinement type 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. Readers are referred to, e.g., Flanagan [2006].)

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 4.18, will show that evaluation produces a result, but not necessarily a value. We note that, unlike some contract calculi—e.g., blame calculus [Wadler and Findler 2009]—function cast applications $\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^{l}\ v$ 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 the semantics, we use evaluation contexts [Felleisen and Hieb 1992] (ranged over by E). 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:

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

where $\mathrm{dom}(\sigma)$ is the domain set of σ and $\mathrm{FV}(\sigma)$ is the set of free term variables in terms and types that appear in the range of 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.

Applying substitutions is almost standard, but for the case for casts. To preserve standard properties of substitution, such as, "applying a substitution to a closed term yields the same term," we keep unused "garbage" bindings out of delayed substitutions, maintaining the invariant 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 how substitutions work, we introduce a few auxiliary notations. For a set S of variables, $\sigma|_S$ denotes the restriction of σ to S. Formally,

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

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

3.1 Definition [Substitution]: Substitution in $\mathrm{F}^\sigma_\mathrm{H}$ is the standard capture-avoiding 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 $\sigma_2 = \sigma(\sigma_1) \uplus (\sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)})$. Here, $\sigma(\sigma_1)$ denotes the (pairwise) composition of σ and σ_1; formally,

$$\sigma(\sigma_1) = (\{x \mapsto \sigma(\sigma_1(x)) \mid x \in \mathrm{dom}(\sigma_1)\}, \{\alpha \mapsto \sigma(\sigma_1(\alpha)) \mid \alpha \in \mathrm{dom}(\sigma_1)\}).$$

Notice that, in the definition of σ_2, the restriction on σ is required to remove garbage bindings. We show that many properties of substitution in lambda calculi hold for our substitution in Appendix A.

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 is given as a small-step relation (Figure 2), split into two sub-relations: one for reductions ($\rightsquigarrow$) and one for subterm reductions and blame lifting ($\longrightarrow$). Rules for these relations are the same as $\mathrm{F_H}$ [Greenberg 2013] except for cast reduction rules. 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 reduces subterms put in 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. We require that $[\![-]\!]$ be "correct" to show type soundness (Section 3.3).

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 5.1.1.) 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 a 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 for type preservation to hold (see Greenberg et al.

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

$$
\begin{array}{lll}
\mathrm{op}\,(v_1,\,\ldots,\,v_n) & \rightsquigarrow\ [\![\mathrm{op}]\!]\,(v_1,\,\ldots,\,v_n) & \text{E\_OP} \\
(\lambda x{:}T_1.\ e_{12})\ v_2 & \rightsquigarrow\ [v_2/x]e_{12} & \text{E\_BETA} \\
(\Lambda\alpha.\ e)\ T & \rightsquigarrow\ [T/\alpha]e & \text{E\_TBETA} \\
\\
\langle T \Rightarrow T\rangle^l_\sigma\ v & \rightsquigarrow\ v & \text{E\_REFL} \\
\langle x{:}T_{11}\rightarrow T_{12} \Rightarrow x{:}T_{21}\rightarrow T_{22}\rangle^l_\sigma\ v \rightsquigarrow & & \text{E\_FUN} \\
\multicolumn{3}{r}{\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)} \\
\multicolumn{3}{r}{\text{when } x{:}T_{11}\rightarrow T_{12} \neq x{:}T_{21}\rightarrow T_{22} \text{ and } x \notin \mathrm{dom}(\sigma) \text{ and}} \\
\multicolumn{3}{r}{y \text{ is fresh and, for } i \in \{1,2\},\ \sigma_i = \sigma|_{\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)) & \text{E\_FORALL} \\
\multicolumn{3}{r}{\text{when } \forall\alpha.\,T_1 \neq \forall\alpha.\,T_2 \text{ and } \alpha \notin \mathrm{dom}(\sigma)} \\
\\
\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l_\sigma\ v & \rightsquigarrow\ \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}\ v & \text{E\_FORGET} \\
\multicolumn{3}{r}{\text{when } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\}} \\
\multicolumn{3}{r}{(\sigma' = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)})} \\
\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma\ v & \rightsquigarrow & \text{E\_PRECHECK} \\
\multicolumn{3}{r}{\langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\ (\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\ v)} \\
\multicolumn{3}{r}{\text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T' \mid e'\}} \\
\multicolumn{3}{r}{(\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2|e_2\})} \text{ and } \sigma_2 = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)})} \\
\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l_\sigma\ v & \rightsquigarrow\ \langle \sigma(\{x{:}T \mid e\}), \sigma([v/x]e), v\rangle^l & \text{E\_CHECK} \\
\\
\langle \{x{:}T \mid e\}, \mathsf{true}, v\rangle^l & \rightsquigarrow\ v & \text{E\_OK} \\
\langle \{x{:}T \mid e\}, \mathsf{false}, v\rangle^l & \rightsquigarrow\ \Uparrow l & \text{E\_FAIL}
\end{array}
$$

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

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

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

[2010] for an explanation). Just like 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. 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. Avoiding this duplication is more efficient and simplifies some of our proofs of parametricity—in particular, we do not need to show that our logical relation is closed under *term* substitution, i.e., two open, logically related terms are related after replacing variables in them with logically related terms. 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 showing preservation: 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, even though T_1 and $[\alpha/\alpha]\,T_1$ are semantically equivalent, 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 does not.

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 is not 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 is not 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 do not have to check the left-hand side, since that is the source type we are casting *from*.) If the check succeeds, then the active check evaluates to the checked value (E_OK); otherwise, it raises the uncatchable exception $\Uparrow l$ (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:

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

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

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

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:

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

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

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

Second, when a function or universal type is cast into a refinement of a *different* function or universal type:

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

E_REFL is necessary for simple cases, like $\langle \mathsf{Int} \Rightarrow \mathsf{Int}\rangle^l\, 5 \longrightarrow 5$. Hopefully, such a useless 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 5.1.1.)

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 $\mathcal{C}^l_\sigma(T_1, T_2, v)$:

$$\begin{array}{rcl}
\mathcal{C}^l_\sigma(T, T, v) & = & v \\
\mathcal{C}^l_\sigma(\{x{:}T_1 \mid e\}, T_2, v) & = & \mathcal{C}^l_\sigma(T_1, T_2, v) \\
\mathcal{C}^l_\sigma(T_1, \{x{:}T_2 \mid e\}, v) & = & \mathsf{let}\ x = \mathcal{C}^l_\sigma(T_1, T_2, v)\ \mathsf{in}\ \langle \sigma(\{x{:}T_2 \mid e\}), \sigma(e), x\rangle^l \\
 & & \hfill \text{(where } x \notin \mathrm{dom}(\sigma)) \\
\mathcal{C}^l_\sigma(\forall\alpha.T_1, \forall\alpha.T_2, v) & = & \Lambda\alpha.\ \mathcal{C}^l_\sigma([\alpha/\alpha]T_1, T_2, v\ \alpha)\ \ \text{(where } \alpha \notin \mathrm{dom}(\sigma)) \\
\mathcal{C}^l_\sigma(x{:}T_{11} \to T_{12}, x{:}T_{21} \to T_{22}, v) & = & \\
\multicolumn{3}{l}{\lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ y = \mathcal{C}^l_\sigma(T_{21}, T_{11}, x)\ \mathsf{in}\ \mathcal{C}^l_\sigma([y/x]T_{12}, T_{22}, v\ y) \text{(where } x, y \notin \mathrm{dom}(\sigma))}
\end{array}$$

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 us consider the cast $\langle T_1 \Rightarrow T_2\rangle^l\ v$, where we omit the delayed substitution for brevity. 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 does not work, we will reduce by E_FORGET until the left-hand side does not have any refinements—possibly zero steps, when the source type T_1 is already unrefined. As E_FORGET applies, either: (a) we strip away all of the refinements; (b) we are casting from T to $\mathrm{unref}_n(T)$, and after n steps E_REFL eventually applies and the entire cast disappears; or (c) at some point we can apply E_CHECK, and the cast disappears. Assuming E_REFL and E_CHECK do not apply, we eventually reduce to $\langle \mathrm{unref}(T_1) \Rightarrow T_2\rangle^l\ v$. Next, we apply E_PRECHECK until the cast is completely decomposed into one-step casts of the form $\langle T' \Rightarrow \{x{:}T' \mid e'\}\rangle^l$. If T_2 has no refinements, we will just apply one of the structural rules, like E_REFL, E_FUN, or E_FORALL. If it does have refinements, though, then we will get:

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

where n is one less than the number of refinements on T_2. At this point, there remain 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 cannot do anything: any value it applies must have already had type $\sigma(T)$, so what could go wrong during any of the ensuing 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][7]. We have not been able to prove parametricity for a system

[7]The upcast lemma in Belo et al. [2011] is for a system with E_REFL, where a reflexive cast is trivially equivalent to the identity function.

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 $\boxed{\Gamma \vdash T}$

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Typing rules for $\mathrm{F}_{\mathrm{H}}^{\sigma}$. The rules marked $*$ are for "runtime" terms.

without E_REFL because our logical relation does not require terms to be well typed; see Section 5.1.1.

3.3. Static typing

The type system comprises three mutually recursive judgments (Figure 3): context well formedness ($\vdash \Gamma$), type well formedness ($\Gamma \vdash T$), and term typing ($\Gamma \vdash e : T$).

The typing rules are the same as F_H [Greenberg 2013] except for T_CAST, the typing rule for casts. The rules for contexts and types are unsurprising. The rules for terms are mostly standard. First, the T_CONST and T_OP rules use the ty function to assign well-formed, closed (possibly dependent) monomorphic first-order types to constants and operations, respectively. We require (a) that constants belong to $\mathcal{K}_{\mathrm{unref}(\mathsf{ty}(k))}$ and satisfy the predicate (if any) of $\mathsf{ty}(k)$, and (b) that $[\![\mathrm{op}]\!]$ 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 is imperative that a term like

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

is not 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 that appear in the range of a delayed substitution in a direct way—though well typed programs will start with and preserve well typed substitutions. Finally, it is critical that compatibility is substitutive, i.e., that if $T_1 \parallel T_2$, then $([e/x]\,T_1) \parallel T_2$ (Lemma A.28).

Some of the typing rules—T_CHECK, T_BLAME, T_EXACT, T_FORGET, and T_CONV—are “runtime only.” These rules are not 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 do not 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 should not *start* with failures. Finally, we exclude T_FORGET because we imagine that source programs have all type changes explicitly managed by casts. The conclusions of these rules use a context Γ, but all terms and types in premises have to be well typed and well formed under the empty context. Even though runtime terms and their typing rules should only ever occur in the 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 assign blame any well formed, closed type—doing so 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 ensures that e_2 is an intermediate state during checking $[v/x]e_1$. 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. T_EXACT is a neatly extensional, syntactic, and subtyping-free replacement for the technique using selfified types and subtyping [Ou et al. 2004].

Finally, the T_CONV rule is motivated by the requirement that terms of $[e_1/x]\,T$ and $[e_2/x]\,T$ should be able to be typed at both types if $e_1 \longrightarrow e_2$—it is necessary to

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

prove preservation; see also the discussion in Section 2.3. These types are *convertible* in $\mathrm{F}_{\mathrm{H}}^{\sigma}$ and T_CONV allows terms to be retyped at convertible types. To determine which types are convertible, we define a conversion relation $\equiv$, which we also call *common-subexpression reduction*, or CSR [Greenberg 2013], 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 $e_1 \longrightarrow^* e_2$; the fact that the substituted terms are related by reduction is the reason why $\equiv$ is called CSR. The rules shown in Figure 4 are the same in Greenberg [2013]. 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 which appears in the range of σ_1 reduces to one of σ_2. We remark that this conversion relation is different from that given in the prior work by Belo et al. [2011],[8] where their conversion relation is defined in terms of parallel reduction. As discussed in Section 2.4, however, it turns out that their conversion relation is flawed. 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 (Theorem A.41, when a term typed by T_APP steps by E_REDUCE/E_REFL).

[8]Actually, the paper omits a formal definition, which appears in Greenberg [2013].

4. PROPERTIES OF $\mathrm{F}_{\mathrm{H}}^{\sigma}$

We show that well-typed programs do not get stuck—a well typed term evaluates to a result, i.e., a value or a blame (if evaluation terminates at all)[9]—via progress and preservation [Wright and Felleisen 1994].

As Greenberg [2013] and Sekiyama et al. [2015] have pointed out, the "value inversion" lemma (Lemma 4.6), which says values typed at refinement types must satisfy their refinements, is a critical component of any sound manifest contract system with constants and operations, especially for proving progress, because operations require their arguments to satisfy refinements of their domain types (see Section 3.3). The type soundness proof in Belo et al. [2011] is missing this lemma—and can never have it, due to the flawed conversion relation. Greenberg [2013] leaves a property which the value inversion depends on as a conjecture—which turns out to be false. This value inversion lemma is not merely a technical device to prove progress. Together with progress and preservation, 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. From this insight, we can find that a manifest contract system satisfying the value inversion lemma is consistent in the sense of Section 2.4 because, for example, $\{x{:}T \mid \mathsf{false}\}$ has no well typed cast-free terminating expressions for any type T. Conversely, a manifest calculus with an inconsistent contract system would not have type soundness for the lack of the value inversion lemma (at least if it provides constants and operations).

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 4.4), which says that common-subexpression reduction does not change the behavior of terms. Finally, using these properties, we show progress (Theorem 4.15) and preservation (Theorem 4.17), which imply type soundness (Theorem 4.18). In this section, we only give statements of the main lemmas and theorems; proofs are in Appendix A.

4.1. Cotermination

First, we show cotermination, the foundation for both type soundness and parametricity. We start with cotermination in the simplest situation, namely, where substitutions map only one term variable, and then we show general cases. The key observation in proving cotermination is that the relation $\{([e_1/x]e, [e_2/x]e) \mid e_1 \longrightarrow e_2\}$ is a weak bisimulation (Lemmas 4.1 and 4.2).

4.1 Lemma [Weak bisimulation, left side (Lemma A.11)]: 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 $e' = [e_1/x]e''$.

4.2 Lemma [Weak bisimulation, right side (Lemma A.14)]: 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 $e' = [e_2/x]e''$.

4.3 Lemma [Cotermination, one variable (Lemma A.15)]: 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 $[e_1/x]e \longrightarrow^* \mathsf{true}$.

4.4 Lemma [Cotermination]: (Lemma A.16) Suppose that $\sigma_1 \longrightarrow^* \sigma_2$.

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

---

[9] In fact, $\mathrm{F}_{\mathrm{H}}^{\sigma}$ is terminating, as we will discover in Section 5.

PROOF. By induction on the size of $\mathsf{dom}(\sigma_1)$ with Lemma 4.3. $\square$

4.2. Type soundness

Using cotermination, we show value inversion and then type soundness in a standard syntactic way, starting with various substitution lemmas.

4.5 Lemma [Cotermination of refinement types (Lemma A.17)]: 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.

4.6 Lemma [Value inversion (Lemma A.18)]: If $\emptyset \vdash v : T$ and $\mathrm{unref}_n(T) = \{x{:}T_n \mid e_n\}$ then $[v/x]e_n \longrightarrow^* \mathsf{true}$.

We use unref here to ensure that the value satisfies *all* of the predicates in its (possibly nested) refinement type.

4.7 Lemma [Term substitutivity of conversion (Lemma A.24)]:
If $T_1 \equiv T_2$ and $e_1 \longrightarrow^* e_2$ then $[e_1/x]T_1 \equiv [e_2/x]T_2$.

4.8 Lemma [Type substitutivity of conversion (Lemma A.25)]:
If $T_1 \equiv T_2$ then $[T/\alpha]T_1 \equiv [T/\alpha]T_2$.

4.9 Lemma [Term weakening (Lemma A.31)]: 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 $\vdash \Gamma, x{:}T', \Gamma'$.

4.10 Lemma [Type weakening (Lemma A.32)]: 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 $\vdash \Gamma, \alpha, \Gamma'$.

4.11 Lemma [Term substitution (Lemma A.33)]: If $\Gamma \vdash e' : T'$, then

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

4.12 Lemma [Type substitution (Lemma A.34)]: If $\Gamma \vdash T'$ then

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

As is standard for type systems with conversion rules, we must prove inversion lemmas 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.3.

4.13 Lemma [Lambda inversion (Lemma A.35)]: If $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$, then there exists some T_2 such that

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

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.

4.14 Lemma [Canonical forms (Lemma A.38)]: If $\emptyset \vdash v : T$, then:

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

4.15 Theorem [Progress (Theorem A.39)]: 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 has the well formed, closed type of the term. With regularity, we can immediately know that the original type is well formed.

4.16 Lemma [Regularity (Lemma A.40)]: (1) If $\Gamma \vdash e : T$, then $\vdash \Gamma$ and $\Gamma \vdash T$; and (2) if $\Gamma \vdash T$ then $\vdash \Gamma$.

4.17 Theorem [Preservation (Theorem A.41)]: If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.

4.18 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 4.15 and 4.17, and the second is by $\emptyset \vdash v : T$ and Lemma 4.6. □

5. PARAMETRICITY

We prove relational parametricity for three reasons: (1) it yields powerful reasoning techniques such as free theorems [Reynolds 1983; Wadler 1989] and the upcast lemma [Belo et al. 2011]; (2) it indicates that contracts do not interfere with type abstraction, i.e., that $\mathrm{F}_{\mathrm{H}}^{\sigma}$ 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—except that our logical relation includes not only well-typed terms and well-formed types but also ill-typed terms and ill-formed types.

5.1. Logical relation

We begin by defining two relations: a *result relation* $r_1 \sim r_2 : T;\theta;\delta$ relating closed results, defined as the least fixpoint on the type index T; and a *term relation* $e_1 \simeq e_2 : T;\theta;\delta$ relating closed expressions, which must evaluate to results in the first relation. (These results and expressions are *not* necessarily well typed. See the discussion below.) The definitions are shown in Figure 5.[10] Both relations have three indices: a (possibly open) type T, a substitution θ for type variables, and a substitution δ for term variables. Type substitutions θ, which give the interpretation of free type variables in T, map type variables α to triples (R, T_1, T_2) comprising a binary relation R on closed results and two closed types T_1 and T_2, to be used as the concrete substitution of α on the left- and right-hand terms. (The results in R and the two types T_1 and T_2 *do not*

[10]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 results and terms $\boxed{r_1 \sim r_2 : T; \theta; \delta}$ $\boxed{e_1 \simeq e_2 : T; \theta; \delta}$

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

$$
e_1 \simeq e_2 : T; \theta; \delta \Longleftrightarrow \exists r_1 r_2, e_1 \longrightarrow^* r_1 \wedge e_2 \longrightarrow^* r_2 \wedge r_1 \sim r_2 : T; \theta; \delta
$$

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

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

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

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

Fig. 5. The logical relation for parametricity

have to be well typed/formed.) Term substitutions δ map variables to pairs of closed (not necessarily well typed) 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 $\theta(\alpha) = (R, T_1, T_2)$. We also use the following notations:

$$
\begin{array}{lll}
\theta[\alpha \mapsto R, T_1, T_2] & = \theta \cup \{\alpha \mapsto R, T_1, T_2\} & \text{if } \alpha \notin \mathrm{dom}(\theta) \\
\delta[(v_1, v_2)/x] & = \delta \cup \{x \mapsto v_1, v_2\} & \text{if } x \notin \mathrm{dom}(\delta)
\end{array}
$$

With these definitions out of the way, the result 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 we 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 $\simeq$ relation on closed terms in a straightforward manner: terms are related if and only if they both evaluate to related results. Divergent terms are not related to each other—though we will discover that divergent well typed terms do not exist in $\mathrm{F}_{\mathrm{H}}^{\sigma}$. 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 results 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 θ and δ. 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 types, 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 not necessarily well typed, 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 [2000]). We have two motivations for allowing ill typed terms in our relation. 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 which 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 will not necessarily be well typed at the other, whether definitions are left- or right-leaning. Second, this parametricity relation is designed so that a certain kind of casts have no effect, as Belo et al. [2011] attempt. Ultimately, we would like to define a subtype relation $T_1 <: T_2$, and show what we call upcast lemma that, 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 do not 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 of the result relation. 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.

5.1.1. Cast reflexivity. 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 uninstantiated 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 (modulo blame). 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.

5.2. Parametricity

Now we can set about proving parametricity. The proof of parametricity (Theorem 5.5) of $\mathrm{F}_{\mathrm{H}}^{\sigma}$ is trickier than that of the standard polymorphic lambda calculus, due to (1) dependent functions, (2) type convertibility, and (3) casts. Before stating parametricity, we discuss these issues; see Appendix A for the proofs of it and lemmas.

In $\mathrm{F}_{\mathrm{H}}^{\sigma}$, it is not as easy as in System F to show that a well-typed term application is logically related to itself due to dependent function types. To see why, consider the application $v_1\, v_2$ such that v_1 and v_2 are typed at $x{:}T_1 \to T_2$ and T_1, respectively. Parametricity states that, if v_1 and v_2 are logically related to themselves with θ and δ, respectively, then so is $v_1\, v_2$ at $[v_2/x]\, T_2$. The definition of the logical relation, however, states only that $v_1\, v_2$ are logically related to T_2, not $[v_2/x]\, T_2$, with θ and $\delta[(v_2, v_2)/x]$. Fortunately, as expected, these are equivalent: $v_1\, v_2$ are logically related to itself at $[v_2/x]\, T_2$ with θ and δ iff $v_1\, v_2$ are logically related to itself at T_2 with θ and $\delta[(v_2, v_2)/x]$. Term compositionality stated below generalizes this.

5.1 Lemma [Term compositionality (Lemma A.42)]: If $\theta_1(\delta_1(e)) \longrightarrow^* v_1$ and $\theta_2(\delta_2(e)) \longrightarrow^* v_2$ then $r_1 \sim r_2 : T; \theta; \delta[(v_1, v_2)/x]$ iff $r_1 \sim r_2 : [e/x]\, T; \theta; \delta$.

For a similar reason, we show type compositionality, which is used in other proofs (e.g., Pitts [2000]). In what follows, we write $R_{T,\theta,\delta}$ for $\{(r_1, r_2) \mid r_1 \sim r_2 : T; \theta; \delta\}$.

5.2 Lemma [Type compositionality (Lemma A.45)]:
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ iff $r_1 \sim r_2 : [T'/\alpha]\, T; \theta; \delta$.

For the typing rule T_CONV with type convertibility, we have to show that terms are logically related to themselves at convertible types.

5.3 Lemma [Convertibility (Lemma A.46)]: If $T_1 \equiv T_2$ then $r_1 \sim r_2 : T_1; \theta; \delta$ iff $r_1 \sim r_2 : T_2; \theta; \delta$.

Showing that casts are logically related to themselves is the most cumbersome case in the proof of parametricity. We prove it by induction on 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 *does not depend on delayed substitutions*. It may be easier for the reader

Complexity of casts

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

Fig. 6. Complexity of casts

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. The cc is well defined though the case for casts between dependent function types chooses an *arbitrary* fresh variable, because, for any variable y and z, $cc(\langle [y/x]\,T_1 \Rightarrow T_2 \rangle^l) = cc(\langle [z/x]\,T_1 \Rightarrow T_2 \rangle^l)$ if y and z do not occur free in T_1 and T_2.

5.4 Lemma [Cast reflexivity (Lemma A.47)]: 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.

5.5 Theorem [Parametricity (Theorem A.48)]: (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 have that logically related programs are by definition *behaviorally equivalent*: if $\emptyset \vdash e_1 \simeq e_2 : B$, then e_1 and e_2 coterminate at equal results. Ideally, logically related terms are also contextually equivalent and vice versa. In fact, to show correctness of static contract checking with respect to the *semantics*, we need this coincidence in addition to the Upcast Lemma because the Upcast Lemma says that an upcast and an identity function are logically related but not that they are contextually equivalent. Going further than Belo et al. [2011] and Greenberg [2013] and proving that the logical relation and contextual equivalence coincide is left as future work.

6. THREE VERSIONS OF F_H

We compare F_H^σ with two prior formulations of F_H without delayed substitution: Belo et al. [2011] and Greenberg's thesis [Greenberg 2013]. Both of these define variants of F_H, claiming type soundness, parametricity, and upcast elimination. All of these results depend on two properties of the F_H type conversion relation: substitutivity (Lemma 4.7) and cotermination (Lemma 4.4).

6.1. 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 $\Rrightarrow$ as the conversion relation (parallel reduction is reflexive by definition); we show selected rules of $\Rrightarrow$ in Figure 7; the full definition is given in Greenberg [2013]. 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 proving type soundness of F_H. Unfor-

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

$$\frac{v_i \Rrightarrow v_i'}{\mathrm{op}\,(v_1,\,\ldots,v_n) \Rrightarrow [\![\mathrm{op}]\!]\,(v_1',\,\ldots,v_n')}\ \textsc{EP\_ROp} \qquad \frac{v \Rrightarrow v'}{\langle T \Rightarrow T\rangle^l\, v \Rrightarrow v'}\ \textsc{EP\_RRefl}$$

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

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

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

$$\frac{}{e \Rrightarrow e}\ \textsc{EP\_Refl}$$

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

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

Fig. 7. Selected rules of parallel reduction (for open terms).

tunately, 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 cotermination.

Why does not substitutivity hold in $\mathrm{F_H}$, when it did (so easily) in λ_{H}? The trouble is 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 can change which reduction rules apply—both counterexamples to substitutivity in Figure 8 exploit these flaws.

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 are not performed under another, related substitution (counterexample 3 in Figure 8).

6.2. $\mathrm{F_H}$ 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 $\mathrm{F_H^\sigma}$'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 ill-typed terms that do not 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 do not have this problem, but we need our conversion relation to prove progress and preservation—we cannot use arguments about typing in our proof of cotermination. In short, Greenberg's Conjecture

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.

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

Observe that $e_1' \Rrightarrow e_2'$ (by EP_REFL) and $e_1 \Rrightarrow e_2$ (by EP_RFUN). We have $[5/x]e_1 = \langle T_1 \to [5/x]T_2 \Rightarrow T_1 \to [5/x]T_2\rangle^l\, v \Rrightarrow [5/x]v$ 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 $\mathrm{F_H}$

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

3.2.1 on page 88 is false; it seems that the evaluation relation is defined in such a way that substitutions can affect which cast reduction rules are chosen.

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

Our calculus, $\mathrm{F}_{\mathrm{H}}^{\sigma}$, can statically determine which cast reduction rule is chosen thanks to our definition of substitution (Definition 3.1). In Lemma 4.4, 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.

6.4. Discussion

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.

Introducing explicit tagging [Wadler and Findler 2009] is an attractive alternative approach. In an explicitly tagged manifest contract system, the only values inhabiting refinement types are tagged as such, e.g., $v_{\{x:T|e\}}$; the operational semantics then manages tags on values, tagging in E_OK 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 4.6). Such a semantics would need to get stuck when casts are applied to inappropriately tagged arguments, since typing cannot be used in the proof of cotermination. Explicit tagging has not yet been tried in a setting with dependent types; it is not entirely clear how to handle substitution and type conversion.

Another option would be to perform type checking in stages: removing refinements from an $\mathrm{F}_{\mathrm{H}}^{\sigma}$ term should always yield a well typed System F term. By only considering System F-typeable terms, we might be able to rule out the counterexample to cotermination. It is not clear whether this would reduce the total proof burden, though.

Finally: what kind of calculus *would not* 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 do not affect reduction—the intuition underlying our proof of cotermination.

7. 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, with both static verification and dynamic checking, in greater detail.

7.1. 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 Morris [1973] and, later, Pierce and Sumii [2000]).

Guha et al. [2007] define latent contracts with polymorphic signatures, maintaining abstraction with sealed "coffers"; they do not prove parametricity. Matthews and Ahmed [2008] claim parametricity for a polymorphic multi-language system with a similar policy, though some of the proofs are not correct per Neis et al. [2009], who use dynamic type generation to restore parametricity in the presence of intensional type analysis. F_H^σ's contracts are subordinate to the type system, so the parametricity result does not require dynamic type generation. Ahmed et al. [2009] and Ahmed et al. [2011] define polymorphic calculi for gradual typing [Siek and Taha 2006]; the former uses global runtime seals, while the latter uses local syntactic "barriers" instead. The type bindings in that work inspired the delayed substitution in this one. Neither calculus proves parametricity and, worse, Ahmed et al. [2011] have a flaw—the proof of so-called "Jack-of-All-Trades," a key property to show blame theorem [Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009], is wrong.[11] Takikawa et al. [2012] study gradual typing with *row polymorphism* for object-oriented languages with first-class classes, but without studying parametricity. Moore et al. [2014] develop SHILL, a secure shell scripting language, to control the authority of a shell script (and programs invoked by it) in a declarative, fine-grained way. Security policies in their work are described and ensured by using contracts. Although the SHILL language supports latent contracts with *bounded polymorphism*, Moore et al. do not study parametricity.

It is probably possible to combine F_H^σ with the barrier calculus of Ahmed et al., yielding a polymorphic blame calculus [Wadler and Findler 2009]. How to prove parametricity and a blame theorem for such a calculus remains an open question, though.

7.2. Combining static and dynamic checking

This section compares F_H^σ and other work on combination of static and dynamic checking. We start with other manifest calculi and then discuss other related work. A less technical overview of manifest calculi is in Section 2.

Simply typed manifest contract calculi. The simply typed contract calculus λ_H is the original foundation of hybrid type checking [Flanagan 2006]. As discussed in Section 2, however, the metatheory of the original λ_H leaves it unclear whether the type system is well defined due to issues with subtyping and monotonicity; subtyping plays an important role in the proof of type soundness. The manifest calculus of Gronski and Flanagan [2007] has the same problem.

Knowles and Flanagan [2010] and Greenberg, Pierce, and Weirich [2010] have revised the original λ_H to resolve the flaw by giving the denotations of types as another source of "well-typed" values; we write Knowles and Flanagan's λ_H KF and Greenberg et al.'s λ_H GPW. Both KF and GPW define syntactic term models of types to use as a source of values in subtyping, though the specifics differ. After adding subtyping and denotational semantics, the type systems of both KF and GPW are well defined. Moreover, as a key property of their calculi, they proved semantic soundness theorems (we write $[\![T]\!]$ for the denotations of type T):

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

in particular

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

This theorem is sufficient for establishing the type soundness of GPW whereas insufficient for KF—due to different definitions of $[\![-]\!]$—and so Knowles and Flanagan prove "syntactic" type soundness on top of their semantic foundation. Although these calculi have been *proven* to be sound, the situation in KF and GPW is somewhat unsatisfying: having to use semantic techniques throughout makes adding some features—polymorphism, state and other effects, concurrency—difficult. For example, a semantic

[11] See https://plt.eecs.northwestern.edu/blame-for-all/blame-for-all.pdf

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? To avoid such a sad situation, Belo et al. propose a syntactic construction of manifest calculi but there are technical flaws in their calculus (Sections 2.3 and 6.1).

The metatheory of $\mathrm{F}_{\mathrm{H}}^{\sigma}$ is entirely syntactic and correct. Similarly to F_{H}, it solves the problem by avoiding subtyping—which is what forced the circularity and denotational semantics in the first place—and introducing T_EXACT, T_CONV, and convertibility $\equiv$ instead. The T_EXACT rule:

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

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 nonempty contexts there, then we would need to apply a closing substitution to $[v/x]e$ before checking if it reduces to true... leading to the same issues with closing substitutions earlier work has suffered from. As for T_CONV and convertibility, convertibility is much simpler than GPW and Belo et al. [2011]. It does not, unfortunately, completely simplify the proof: we must prove that our conversion relation is a weak bisimulation to establish *cotermination* (Lemma 4.4) before proving type soundness.

The SAGE *language.* Gronski et al. [2006] develop the SAGE language, which supports subsumption for subtyping, casts, general refinements, polymorphism, recursive functions, recursive types, the Dynamic type, the Type:Type discipline. SAGE avoids the circularity of Flanagan's λ_{H}, changing formalization of subtyping: in SAGE, $\{x{:}T \mid e_1\}$ is a subtype of $\{x{:}T \mid e_2\}$ if a theorem prover can prove the implication from e_1 to e_2. Since the theorem prover is independent of SAGE, the type system is well defined. Naturally, the metatheory of SAGE rests on the theorem prover. SAGE states axioms strong enough to show type soundness—for example, it requires the prover to be able to show $[e_1/x]e$ evaluates to true iff $[e_2/x]e$ does when $e_1 \longrightarrow e_2$, which works similarly to cotermination in $\mathrm{F}_{\mathrm{H}}^{\sigma}$. Although Gronski et al. have shown type soundness of SAGE, they do not deal with parametricity, while we show it in $\mathrm{F}_{\mathrm{H}}^{\sigma}$. In fact, it is difficult to show parametricity in calculi with recursive functions [Pitts 2000], recursive types [Ahmed 2006], the Dynamic type [Matthews and Ahmed 2008], and/or Type:Type. In addition, axiomatization of theorem provers could bring us to an unsatisfactory situation. For example, the axiom system of Gronski et al. is inconsistent, though fixed by Knowles [2014].

A manifest calculus with algebraic datatypes. Sekiyama et al. [2015] introduce a manifest calculus with algebraic datatypes and show conjecture-free type soundness of their calculus, although their calculus does not support polymorphism. The metatheory of it rests on CSR, while they do not adopt delayed substitutions. Instead, in order to ensure that how cast reduces is determined statically (this is crucial for showing cotermination), they drop cast reduction rules that see syntactic equality of the source and target types in a cast, like E_REFL of $\mathrm{F}_{\mathrm{H}}^{\sigma}$—*any* cast in their calculus works as follows: it will first drop all refinements in the source type, apply a structural cast, and then check all refinements in the target type. Although their calculus does not need delayed substitutions, it is not clear that parametricity holds in manifest calculi with such cast semantics because of E_REFL's important role in showing parametricity in $\mathrm{F}_{\mathrm{H}}^{\sigma}$.

Dependent types with dynamic typing. Ou et al. [2004] study integration of certified and uncertified program fragments—all refinements in certified parts are checked statically whereas all those in uncertified parts are checked at runtime. They model static checking as subtyping checking and dynamic checking as compilation to predicate checking with `if`-expressions. Their calculus deal with the issues of preservation by supporting a special typing rule to assign "selfified" types to terms and subsumption for subtyping. Unlike manifest calculi, they restrict refinements (and so also arguments to dependent functions) to be syntactically pure in order to make static checking decidable. They also axiomatize requirements on theorem provers, like Gronski et al. [2006].

Static analysis and verification using path information. Much work on static program analysis and verification (e.g., Hoare [1969]; Paulin-Mohring [1993]; Xi et al. [2003]; Cheney and Hinze [2003]; Nanevski et al. [2006]; Rondon et al. [2008]; Kawaguchi et al. [2009]; Knowles and Flanagan [2009]; Bierman et al. [2010]; Tobin-Hochstadt and Felleisen [2010]; Chugh et al. [2012]; Nguyen et al. [2014]) employs path information of conditional expressions—for example, when `if`-expressions are verified, the conditional expressions are supposed to hold in then-expressions whereas they are not to hold in else-expressions. Kent et al. [2016] is particularly related, augmenting a path-sensitive type system with refinements; they extract a smaller logical language for predicates, while our refinements use arbitrary code. In any case, path information can be thought as "dynamic" because it is a result of an analysis of what values are examined at runtime. Although $\mathrm{F}_{\mathrm{H}}^{\sigma}$ does not keep track of path information directly, we can simulate by encoding an if-expression ($\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$) in source programs as:

$$\begin{array}{l} \mathsf{if}\ e_1\ \mathsf{then}\ \ (\mathsf{let}\ x = \langle \mathsf{Bool} \Rightarrow \{y{:}\mathsf{Bool} \mid e_1\}\rangle^l\ \mathsf{true}\ \mathsf{in}\ e_2) \\ \qquad\quad \mathsf{else}\ \ (\mathsf{let}\ x = \langle \mathsf{Bool} \Rightarrow \{y{:}\mathsf{Bool} \mid \mathsf{not}\ e_1\}\rangle^l\ \mathsf{true}\ \mathsf{in}\ e_3) \end{array}$$

where x and y are fresh. Under this encoding, e_2 and e_3 are typed under a binding that x is given type $\{y{:}\mathsf{Bool} \mid e_1\}$ and $\{y{:}\mathsf{Bool} \mid \mathsf{not}\ e_1\}$, respectively. This corresponds to a path-sensitive typing rule for if-expressions, found, e.g., in Rondon et al. [2008]:

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma, e_1 \vdash e_2 : T \quad \Gamma, \mathsf{not}\ e_1 \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

(A Boolean expression e in a context intuitively means "e reduces to true.") Such path information would be useful if we consider static verification for manifest contracts.

There is a large body of literature on static checking of refinement types; liquid types is the most relevant recent work [Rondon et al. 2008; Jhala 2014]. Liquid types are entirely static, rejecting programs that cannot be verified; moreover, they draw the predicates in their refinements from a user-specified set, rather than arbitrary code.

8. CONCLUSION

$\mathrm{F}_{\mathrm{H}}^{\sigma}$ combines parametric polymorphism and manifest contracts. When we say "parametrically" polymorphic, we mean in particular that the relation R used to relate terms at type variables in the logical relation is a *parameter* of the logical relation, which admits any instantiation of R.[12] We offer the first conjecture-free, completely correct polymorphic manifest calculus with *general* refinements, where refinements can apply to any type, not just base types.

We hope to extend $\mathrm{F}_{\mathrm{H}}^{\sigma}$ with barriers for dynamically checked polymorphism [Ahmed et al. 2011], and with state. (Though we acknowledge that state is a difficult open problem; see Greenberg [2015a].) Casts between functions and quantified types explicitly introduce function proxies; can our polymorphic semantics be made space ef-

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

ficient [Herman et al. 2010; Greenberg 2015b]? We also hope that F_H^σ's operational semantics and (relatively) simple type system will help developers implement contracts. Finally, we are curious to see what we can do with a contract language with the reasoning principles derivable from relational parametricity.

We elide subtyping and a proof of the Upcast Lemma, which states that a cast from a subtype to a supertype is logically related to an identity function—we believe those in Belo et al. [2011] and Greenberg et al. [2010] can be adapted, since the parametricity relation has not materially changed in F_H^σ. The first two authors are also working on a complete account of another polymorphic manifest contract calculus with recursion, a parametricity relation that has a clear relationship to contextual equivalence, and proofs of subtyping [Sekiyama and Igarashi 2012].

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. Finally, we thank the journal reviewers for their hard work and helpful advice.

REFERENCES

ABADI, M., CARDELLI, L., CURIEN, P.-L., AND LÉVY, J.-J. 1991. Explicit substitutions. *Journal of Functional Programming (JFP) 1,* 4, 375–416.

ABADI, M., CARDELLI, L., PIERCE, B. C., AND PLOTKIN, G. D. 1989. Dynamic typing in a statically-typed language. In *Principles of Programming Languages (POPL)*. 213–227.

AHMED, A. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*. 69–83.

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)*. 201–214.

BARENDREGT, H. P. 1992. Lambda calculi with types. In *Handbook of Logic in Computer Science (Vol. 2)*, S. Abramsky, D. M. Gabbay, and S. E. Maibaum, Eds. Oxford University Press, Inc., 117–309.

BELO, J. F., GREENBERG, M., IGARASHI, A., AND PIERCE, B. C. 2011. Polymorphic contracts. In *European Symposium on Programming (ESOP)*. 18–37.

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)*. 105–116.

CARDELLI, L. 1986. A polymorphic λ-calculus with Type:Type. Technical Report 10, DEC Systems Research Center, Palo Alto, CA.

CHENEY, J. AND HINZE, R. 2003. First-class phantom types. Technical report, Cornell University.

CHUGH, R., HERMAN, D., AND JHALA, R. 2012. Dependent types for JavaScript. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 587–606.

DE BRUIJIN, N. G. 1980. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, Eds. Academic Press, 579–606.

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)*. 48–59.

FLANAGAN, C. 2006. Hybrid type checking. In *Principles of Programming Languages (POPL)*. 245–256.

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

FREEMAN, T. AND PFENNING, F. 1991. Refinement types for ML. In *Programming Language Design and Implementation (PLDI)*. 268–277.

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

GREENBERG, M. 2015a. Combining manifest contracts with state. HOPE.

GREENBERG, M. 2015b. Space-efficient manifest contracts. In *Principles of Programming Languages (POPL)*. 181–194.

GREENBERG, M., PIERCE, B. C., AND WEIRICH, S. 2010. Contracts made manifest. In *Principles of Programming Languages (POPL)*. 353–364.

GRONSKI, J. AND FLANAGAN, C. 2007. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*. 54–70.

GRONSKI, J., KNOWLES, K., TOMB, A., FREUND, S. N., AND FLANAGAN, C. 2006. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*.

GROSSMAN, D., MORRISETT, G., AND ZDANCEWIC, S. 2000. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS) 22,* 6, 1037–1080.

GUHA, A., MATTHEWS, J., FINDLER, R. B., AND KRISHNAMURTHI, S. 2007. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*. 29–40.

HARPER, R., HONSELL, F., AND PLOTKIN, G. 1993. A framework for defining logics. *Journal of the ACM (JACM) 40,* 1, 143–184.

HENGLEIN, F. 1992. Dynamic typing. In *European Symposium on Programming (ESOP)*. 233–253.

HERMAN, D., TOMB, A., AND FLANAGAN, C. 2010. Space-efficient gradual typing. *Higher Order Symbol. Comput. 23,* 2, 167–189.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10, 576–580.

JHALA, R. 2014. Refinement types for haskell. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification*. PLPV '14. ACM, New York, NY, USA, 27–27.

KAWAGUCHI, M., RONDON, P. M., AND JHALA, R. 2009. Type-based data structure verification. In *Programming Language Design and Implementation (PLDI)*. 304–315.

KENT, A. M., KEMPE, D., AND TOBIN-HOCHSTADT, S. 2016. Occurrence typing modulo theories. In *Programming Language Design and Implementation (PLDI)*. 296–309.

KNOWLES, K. 2014. Executable refinement types. Ph.D. thesis, University of California, Santa Cruz.

KNOWLES, K. AND FLANAGAN, C. 2009. Compositional reasoning and decidable checking for dependent contract types. In *Programming Languages meets Program Verification (PLPV)*. 27–38.

KNOWLES, K. AND FLANAGAN, C. 2010. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS) 32*, 6:1–6:34.

MANDELBAUM, Y., WALKER, D., AND HARPER, R. 2003. An effective theory of type refinements. In *International Conference on Functional Programming (ICFP)*. 213–225.

MATTHEWS, J. AND AHMED, A. 2008. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*. 16–31.

MITCHELL, J. C. AND PLOTKIN, G. D. 1985. Abstract types have existential type. In *Principles of Programming Languages (POPL)*. 37–51.

MOORE, S., DIMOULAS, C., KING, D., AND CHONG, S. 2014. SHILL: A secure shell scripting language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 183–199.

MORRIS, JR., J. H. 1973. Types are not sets. In *Principles of Programming Languages (POPL)*. 120–124.

NANEVSKI, A., MORRISETT, G., AND BIRKEDAL, L. 2006. Polymorphism and separation in Hoare type theory. In *International Conference on Functional Programming (ICFP)*. 62–73.

NEIS, G., DREYER, D., AND ROSSBERG, A. 2009. Non-parametric parametricity. In *International Conference on Functional Programming (ICFP)*. 135–148.

NGUYEN, P. C., TOBIN-HOCHSTADT, S., AND VAN HORN, D. 2014. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. ACM, New York, NY, USA, 139–152.

OU, X., TAN, G., MANDELBAUM, Y., AND WALKER, D. 2004. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science (TCS)*. 437–450.

PAULIN-MOHRING, C. 1993. Inductive definitions in the system Coq - rules and properties. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*. 328–345.

PIERCE, B. AND SUMII, E. 2000. Relating cryptography and polymorphism. Unpublished manuscript. http://www.cis.upenn.edu/~bcpierce/papers/infohide.ps.

PITTS, A. M. 2000. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10,* 3, 321–359.

REYNOLDS, J. C. 1983. Types, abstraction and parametric polymorphism. In *IFIP Congress*. 513–523.

RONDON, P. M., KAWAGUCHI, M., AND JHALA, R. 2008. Liquid types. In *Programming Language Design and Implementation (PLDI)*. 159–169.

SEKIYAMA, T. AND IGARASHI, A. 2012. Logical relations for a manifest contract calculus, fixed. http://hope2012.mpi-sws.org/. Talk abstract and slides.

SEKIYAMA, T., NISHIDA, Y., AND IGARASHI, A. 2015. Manifest contracts for datatypes. In *Principles of Programming Languages (POPL)*. 195–207.

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

TAKIKAWA, A., STRICKLAND, T. S., DIMOULAS, C., TOBIN-HOCHSTADT, S., AND FELLEISEN, M. 2012. Gradual typing for first-class classes. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 793–810.

TOBIN-HOCHSTADT, S. AND FELLEISEN, M. 2006. Interlanguage migration: From scripts to programs. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)Companion*. 964–974.

TOBIN-HOCHSTADT, S. AND FELLEISEN, M. 2010. Logical types for untyped languages. In *International Conference on Functional Programming (ICFP)*. 117–128.

WADLER, P. 1989. Theorems for free! In *Conference on Functional Programming and Computer Architecture (FPCA)*. 347–359.

WADLER, P. AND FINDLER, R. B. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*. 1–16.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation 115*, 38–94.
XI, H., CHEN, C., AND CHEN, G. 2003. Guarded recursive datatype constructors. In *Principles of Programming Languages (POPL)*. 224–235.
XI, H. AND PFENNING, F. 1999. Dependent types in practical programming. In *Principles of Programming Languages (POPL)*. 214–227.

A. PROOFS

We give the proofs of (syntactic) type soundness (Theorem 4.18) and parametricity (Theorem A.48) of $\mathrm{F}_{\mathrm{H}}^{\sigma}$, without conjectures. We start with proving standard properties about free variables and substitution (Section A.1) because they are nonstandard and slightly tricky. Section A.2 shows cotermination, a key property to ensure that our type conversion relates only types equivalent "semantically" (in particular, Lemma A.17 deals with the case for refinement types). Using cotermination, we show type soundness via progress (Theorem A.39) and preservation (Theorem A.41) in Section A.3. Finally, Section A.4 shows parametricity (Theorem A.48), which also depends on cotermination.

A.1. Properties of substitution

A.1 Lemma [Free Term Variables After Substitution]: Let σ be a substitution.

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

PROOF. By structural induction on e and T. We mention only the case of casts in the first case. We are given $e = \langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma_1}$. Let $\sigma_2 = \sigma(\sigma_1) \uplus \sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)}$. By definition, $\sigma(e) = \langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma_2}$ and

$$\begin{aligned}
& \mathrm{FV}(\sigma(e)) \\
= \ & ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma_2)) \cup \mathrm{FV}(\sigma_2) \\
= \ & ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)}))) \cup \mathrm{FV}(\sigma_2) \\
= \ & ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma))) \cup \mathrm{FV}(\sigma_2).
\end{aligned}$$

We have $\mathrm{FV}(e) = ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma_1)) \cup \mathrm{FV}(\sigma_1)$, and so

$$\mathrm{FV}(e) \setminus \mathrm{dom}(\sigma) = ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma))) \cup (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)).$$

Thus, it suffices to show that

$$\mathrm{FV}(\sigma_2) = (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(e)}).$$

Here, we have $\mathrm{FV}(\sigma_2) = \mathrm{FV}(\sigma(\sigma_1)) \cup \mathrm{FV}(\sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)})$. By the IHs,

$$\begin{aligned}
\mathrm{FV}(\sigma(\sigma_1)) \ &= \bigcup_{x \in \mathrm{dom}(\sigma_1)} \mathrm{FV}(\sigma(\sigma_1(x))) \cup \bigcup_{\alpha \in \mathrm{dom}(\sigma_1)} \mathrm{FV}(\sigma(\sigma_1(\alpha))) \\
&= \bigcup_{x \in \mathrm{dom}(\sigma_1)} ((\mathrm{FV}(\sigma_1(x)) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1(x))})) \cup \\
&\quad \bigcup_{\alpha \in \mathrm{dom}(\sigma_1)} ((\mathrm{FV}(\sigma_1(\alpha)) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1(\alpha))})) \\
&= (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1)}).
\end{aligned}$$

Thus,

$$\mathrm{FV}(\sigma_2) = (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1)}) \cup \mathrm{FV}(\sigma|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma_1)}),$$

and so it suffices to show that

$$\mathrm{FV}(\sigma|_{\mathrm{AFV}(e)}) = \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1)}) \cup \mathrm{FV}(\sigma|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma_1)}).$$

Since $\mathrm{AFV}(e) = ((\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)) \cup \mathrm{AFV}(\sigma_1)$, we finish. $\square$

A.2 Lemma [Free Type Variables After Substitution]: Let σ be a substitution.

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

PROOF. Similar to Lemma A.1; by structural induction on e and T. $\square$

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. We mention only the case of casts. We are given $e = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}$. By definition:

$$\begin{aligned}\mathrm{FV}(e) &= ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma')) \cup \mathrm{FV}(\sigma')\\ \mathrm{FTV}(e) &= ((\mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2)) \setminus \mathrm{dom}(\sigma')) \cup \mathrm{FTV}(\sigma')\end{aligned}$$

Since $(\mathrm{FV}(e) \cup \mathrm{FTV}(e)) \cap \mathrm{dom}(\sigma) = \emptyset$, we have:

$$\begin{aligned}\mathrm{dom}(\sigma) \cap ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma')) &= \emptyset\\ \mathrm{dom}(\sigma) \cap ((\mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2)) \setminus \mathrm{dom}(\sigma')) &= \emptyset\end{aligned}$$

Thus, $\sigma(\langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}) = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma(\sigma')}$. Since $(\mathrm{FV}(e) \cup \mathrm{FTV}(e)) \cap \mathrm{dom}(\sigma) = \emptyset$, we have $(\mathrm{FV}(\sigma') \cup \mathrm{FTV}(\sigma')) \cap \mathrm{dom}(\sigma) = \emptyset$, and thus $\sigma(\sigma') = \sigma'$ by the IHs. Thus, $\sigma(\langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}) = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}$. $\square$

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. We mention only the case of casts. We are given $e = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma}$. Let $S_1 = \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)$, $S_2 = \mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2)$, and $S = \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)$. By definition, $\sigma_1(e) = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_1'}$ where $\sigma_1' = \sigma_1(\sigma) \uplus \sigma_1|_{S\setminus\mathrm{dom}(\sigma)}$. Thus, $\sigma_2(\sigma_1(e)) = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_1''}$ where

$$\begin{aligned}\sigma_1'' &= \sigma_2(\sigma_1') \uplus \sigma_2|_{S\setminus\mathrm{dom}(\sigma_1')}\\ &= \sigma_2(\sigma_1(\sigma)) \uplus \sigma_2(\sigma_1)|_{S\setminus\mathrm{dom}(\sigma)} \uplus \sigma_2|_{S\setminus\mathrm{dom}(\sigma_1')}.\end{aligned}$$

Also, we have $\sigma_2(e) = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2'}$ where $\sigma_2' = \sigma_2(\sigma) \uplus \sigma_2|_{S\setminus\mathrm{dom}(\sigma)}$, and $\sigma_2(\sigma_1)(\sigma_2(e)) = \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2''}$ where

$$\begin{aligned}\sigma_2'' &= (\sigma_2(\sigma_1))(\sigma_2') \uplus \sigma_2(\sigma_1)|_{S\setminus\mathrm{dom}(\sigma_2')}\\ &= (\sigma_2(\sigma_1))(\sigma_2(\sigma)) \uplus (\sigma_2(\sigma_1))(\sigma_2)|_{S\setminus\mathrm{dom}(\sigma)} \uplus \sigma_2(\sigma_1)|_{S\setminus\mathrm{dom}(\sigma_2')}.\end{aligned}$$

We show that $\sigma_1'' = \sigma_2''$ as follows.

(1) We have $\sigma_2(\sigma_1(\sigma)) = (\sigma_2(\sigma_1))(\sigma_2(\sigma))$ because, for any $x \in \mathsf{dom}(\sigma)$,

$$\begin{array}{rcll} \sigma_2(\sigma_1(\sigma))(x) & = & \sigma_2(\sigma_1(\sigma(x))) & \\ & = & (\sigma_2(\sigma_1))(\sigma_2(\sigma(x))) & \text{(by the IH)} \\ & = & (\sigma_2(\sigma_1))(\sigma_2(\sigma))(x), & \end{array}$$

and for any $\alpha \in \mathrm{dom}(\sigma)$, $\sigma_2(\sigma_1(\sigma))(\alpha) = (\sigma_2(\sigma_1))(\sigma_2(\sigma))(\alpha)$, which can be proven similarly to term variables by the IH.

(2) We show that $\sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma)} = \sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma_2')}$, that is, we show that

$$\mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma)) = \mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma_2')).$$

Here, we have

$$\begin{array}{rcl} \mathsf{dom}(\sigma_2') & = & \mathrm{dom}(\sigma) \cup (\mathrm{dom}(\sigma_2) \cap (S \setminus \mathrm{dom}(\sigma))) \\ & = & (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2)) \cap (\mathrm{dom}(\sigma) \cup (S \setminus \mathrm{dom}(\sigma))) \\ & = & (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2)) \cap (\mathrm{dom}(\sigma) \cup S) \\ & = & \mathrm{dom}(\sigma) \cup (\mathrm{dom}(\sigma_2) \cap S). \end{array}$$

Thus,

$$\begin{array}{rcl} \mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma_2')) & = & \mathrm{dom}(\sigma_1) \cap (S \setminus (\mathrm{dom}(\sigma) \cup (\mathrm{dom}(\sigma_2) \cap S))) \\ & = & \mathrm{dom}(\sigma_1) \cap (S \setminus (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2))) \\ & = & (\mathrm{dom}(\sigma_1) \cap S) \setminus (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2)) \\ & = & (\mathrm{dom}(\sigma_1) \cap S) \setminus \mathrm{dom}(\sigma) \\ & & \quad \text{(since } \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset\text{)} \\ & = & \mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma)). \end{array}$$

(3) We show that $\sigma_2|_{S \setminus \mathrm{dom}(\sigma_1')} = (\sigma_2(\sigma_1))(\sigma_2)|_{S \setminus \mathrm{dom}(\sigma)}$. Since $\mathrm{AFV}(\sigma_2) \cap \mathrm{dom}(\sigma_1) = \emptyset$, we have $(\sigma_2(\sigma_1))(\sigma_2) = \sigma_2$ by Lemma A.3. Thus, it suffices to show that

$$\mathrm{dom}(\sigma_2) \cap (S \setminus \mathrm{dom}(\sigma_1')) = \mathrm{dom}(\sigma_2) \cap (S \setminus \mathrm{dom}(\sigma)),$$

which can be shown similarly to the above.

□

A.2. Cotermination

The key observation in proving cotermination is that the relation $\{([e_1/x]e, [e_2/x]e) \mid e_1 \longrightarrow e_2\}$ is a weak bisimulation. Lemmas A.11 and A.14 show it for cases that left- and right-hand terms first evaluate, respectively; the cases of term and type applications (without reducible subterms) are shown in Lemmas A.7 and A.9, respectively. We show cotermination in the case that substitutions map only one term variable (Lemma A.15), and then show general cases (Lemma A.16).

Throughout the proof, we implicitly make use of the determinism of the semantics.

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

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

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'. Here 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 $\mathsf{dom}(\sigma)$ are fresh. Let z be a fresh variable and $i, j \in \{1, 2\}$. Moreover, let σ_i be

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

and σ_{ij} be $\sigma_i|_{\mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})}$. 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 $\sigma'_j = \sigma|_{\mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})}$ 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 $S_j = \mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})$,

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

The last equation is derived from the fact that

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

$\underline{e'_1 = \langle T_1 \Rightarrow \{y{:}T_2 \mid e\} \rangle^l_\sigma}$: 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

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

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

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

it suffices to show that $[e_i/x]e' = e''_i$. We can show that $[e_i/x]\sigma'_1 \uplus ([e_i/x]|_{\mathrm{AFV}(\{y{:}T_2|e\}) \setminus \mathrm{dom}(\sigma'_1)}) = \sigma_{i1}$ and and $[e_i/x]\sigma'_2 \uplus ([e_i/x]|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma'_2)}) = \sigma_{i2}$ 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, 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 (evaluation relation is defined over closed terms). □

A.10 Lemma: If $e_1 \longrightarrow^* e_2$, then $E[e_1] \longrightarrow^* E[e_2]$.

PROOF. By induction on the number of evaluation steps of e_1 with E_COMPAT. □

A.11 Lemma [Weak bisimulation, left side]: (Lemma 4.1) 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. Here e_1 is not a value, since $e_1 \longrightarrow e_2$.

$\underline{e = x}$: 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 (recall that the evaluation relation is a relation over closed terms).
$\underline{e = v,\ y \text{ where } x \neq y \text{ or } \Uparrow l}$: 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 Lemma A.10. 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).
$\underline{e = e'_1\ T_2}$: Similar to the case of function application, with Lemma A.9 (1).
$\underline{e = \langle\{y{:}T \mid e'_1\}, e'_2, v\rangle^l}$: Similar to the above.

□

A.12 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.13 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.14 Lemma [Weak bisimulation, right side]: (Lemma 4.2)
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 $[e_2/x]\,\mathrm{op}\,(e'_1, \ldots, e'_n)$ takes a step. By Lemma A.12, $[e_1/x]\,\mathrm{op}\,(e'_1, \ldots, e'_n) \longrightarrow^* [e_1/x]\,\mathrm{op}\,(e''_1, \ldots, e''_2)$ for some $e''_1, \ldots, e''_n$ such that $[e_2/x]\,\mathrm{op}\,(e'_1, \ldots, e'_n) = [e_2/x]\,\mathrm{op}\,(e''_1, \ldots, e''_n)$. 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.12, 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.13. 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.12, 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).
$\underline{e = e'_1\ T_2}$: Similar to the case of function application, with Lemma A.9 (2).
$\underline{e = \langle \{y{:}T \mid e'_1\}, e'_2, v\rangle^l}$: Similari to the above.

□

A.15 Lemma [Cotermination, one variable]: (Lemma 4.3)

(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 Lemmas A.11 and A.14 for the induction steps, respectively.
If $[e_1/x]e = \mathsf{true}$, then either $e = x$ and $e_1 = \mathsf{true}$, or $e = \mathsf{true}$. The former contradicts the assumption that $e_1 \longrightarrow e_2$. Thus, $e = \mathsf{true}$ and so $[e_2/x]e = \mathsf{true}$.
If $[e_2/x]e = \mathsf{true}$, then, by Lemma A.12, there exists e' such that $[e_1/x]e \longrightarrow^* [e_1/x]e'$ and $[e_1/x]e'$ is a value and $[e_2/x]e = [e_2/x]e'$. Similarly to the discussion above, we have $e' = \mathsf{true}$ and so finish.
(2) By induction on the number of evaluation steps of e_1 with the first case.

□

A.16 Lemma [Cotermination]: (Lemma 4.4) Suppose that $\sigma_1 \longrightarrow^* \sigma_2$.

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

PROOF. By induction on the size of $\mathsf{dom}(\sigma_1)$ with Lemma A.15. □

A.3. Type soundness

We show type soundness in a syntactic manner, using progress (Theorem A.39) and preservation (Theorem A.41). Cotermination is used to show value inversion (Lemma A.18), which implies consistency of the contract system of $\mathrm{F}_{\mathrm{H}}^{\sigma}$ and is used to show progress in the case for T_OP. After proving properties of convertibility (Lemmas A.19–A.26) and compatibility (Lemmas A.27–A.30), we show progress and preservation, using standard lemmas: weakening lemmas (Lemmas A.31 and A.32), substitution lemmas (Lemmas A.33 and A.34), inversion lemmas (Lemmas A.35–A.37), and canonical forms lemma (Lemma A.38).

A.17 Lemma [Cotermination of refinement types (Lemma 4.5)]: 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 A.16), 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.18 Lemma [Value inversion (Lemma 4.6)]: 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.17 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.19 Lemma [Reflexivity of conversion]:
$T \equiv T$ for all T.

PROOF. By induction on T. □

A.20 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.21 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.20. □

A.22 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.23 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.22. □

A.24 Lemma [Term substitutivity of conversion (Lemma 4.7)]:
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.25 Lemma [Type substitutivity of conversion (Lemma 4.8)]:
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.19). 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.26 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.27 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.28 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.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.
□

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. □

A.31 Lemma [Term weakening (Lemma 4.9)]: 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 will not bother considering these rules. □

A.32 Lemma [Type weakening (Lemma 4.10)]: 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.31. □

A.33 Lemma [Term substitution (Lemma 4.11)]: If $\Gamma \vdash e' : T'$, then

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

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

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.34 Lemma [Type substitution (Lemma 4.12)]: If $\Gamma \vdash T'$ then

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

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

A.35 Lemma [Lambda inversion (Lemma 4.13)]: If $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$, then there exists some T_2 such that

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

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.19), since $\mathrm{unref}(T) = T = x{:}T_1 \rightarrow T_2$.
(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, $\mathrm{unref}(T') \equiv x{:}T_1 \rightarrow T_2$. By weakening, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $T' \equiv T$, we have $x{:}T_1 \rightarrow T_2 \equiv \mathrm{unref}(T') \equiv \mathrm{unref}(T)$ 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 $x{:}T_1 \rightarrow T_2 \equiv \mathrm{unref}(T')$. 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): $x{:}T_1 \rightarrow T_2 \equiv \mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$.
(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 $x{:}T_1 \rightarrow T_2 \equiv \mathrm{unref}(\{x{:}T \mid e\})$. 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 $x{:}T_1 \rightarrow T_{12} \equiv \mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$. $\square$

A.36 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) $\_{:}\sigma(T_1) \rightarrow \sigma(T_2) \equiv \mathrm{unref}(T)$ (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.35. $\square$

A.37 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.35. $\square$

A.38 Lemma [Canonical forms (Lemma 4.14)]: If $\emptyset \vdash v : T$, then:

(1) If $\mathrm{unref}(T) = B$ then v is $k \in \mathcal{K}_B$ for some k.
(2) If $\mathrm{unref}(T) = x{:}T_1 \rightarrow T_2$ then
  (a) v is $\lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$ for some x, T_1', and e_{12}, or

(b) v is $\langle T_1' \Rightarrow T_2' \rangle_\sigma^l$ and $\sigma(T_1') \equiv T_1$ and $\sigma(T_2') \equiv T_2$ for some T_1', T_2', σ, 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.19).
(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.19).
(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.19).
(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.26) 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.21); 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.26) 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 $T_2'' \equiv T_2$ (Lemmas A.20 and A.21); 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.26).

(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 is 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. □

A.39 Theorem [Progress (Theorem 4.15)]: 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 $\mathsf{ty}(\mathrm{op}) = x_1 : T_1 \to \ldots \to x_n : T_n \to T$. By inversion, $\emptyset \vdash e_i : \sigma(T_i)$. Applying the IH from left to right, each of the e_i either steps or is a result.

Suppose everything to the left of e_i is a value. Then either e_i steps or is a result. If $e_i \longrightarrow e_i'$, then $\mathrm{op}(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow \mathrm{op}(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)$ by E_COMPAT. One the other hand, if e_i is a result, there are two cases. If $e_i = \Uparrow l$, then the original expression steps to $\Uparrow l$ by E_BLAME. If e_i is a value, we can continue this process for each of the operation's arguments. Eventually, all of the operations arguments are values. By value inversion (Lemma A.18), 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): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \to T_2)$. 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, $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$ and $\emptyset \vdash e_2 : T_1$.

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

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

By canonical forms (Lemma A.38) on $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$, there are two cases:

($e_1 = \lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$): In this case, $(\lambda x{:}T_1'.\ e_{12})\ v_2 \longrightarrow [v_2/x]e_{12}$ 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.36). 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 $x{:}T_{21} \to T_{22}$ or $\forall\alpha.\,T_{22}$): Incompatible; contradictory.

($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = B$, then by E_CHECK, $\langle B \Rightarrow \{x{:}B \mid e\}\rangle^l_\sigma\ v_2 \longrightarrow \langle \sigma(\{x{:}B \mid e\}), \sigma([v_2/x]e), v_2\rangle^l$. 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 $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_2'')}$.

($T_1' = \alpha$):

($T_2' = \alpha'$): 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 $x{:}T_{21} \to T_{22}$ or $\forall\alpha.\,T_{22}$): Incompatible; contradictory.

($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = \alpha$, then by E_CHECK, $\langle \alpha \Rightarrow \{x{:}\alpha \mid e\}\rangle^l_\sigma\ v_2 \longrightarrow \langle \sigma(\{x{:}\alpha \mid e\}), \sigma([v_2/x]e), v_2\rangle^l$. 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 $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_2'')}$, 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 $\sigma_i = \sigma|_{\mathrm{AFV}(T_{1i}) \cup \mathrm{AFV}(T_{2i})}$ ($i \in \{1,2\}$), 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 $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_1') \cup \mathrm{AFV}(T_2'')}$, 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 $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_1')\cup\mathrm{AFV}(T_2'')}$, 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 $\sigma' = \sigma|_{\mathrm{AFV}(T_1'')\cup\mathrm{AFV}(T_2')}$, 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 $\sigma' = \sigma|_{\mathrm{AFV}(T_1'')\cup\mathrm{AFV}(\{x:T_2''|e_2'\})}$, 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.38). 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 is typed at Bool. If it is 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.40 Lemma [Regularity (Lemma 4.16)]: (1) If $\Gamma \vdash e : T$, then $\vdash \Gamma$ and $\Gamma \vdash T$; and (2) if $\Gamma \vdash T$ then $\vdash \Gamma$.

PROOF. By induction on the typing and well formedness derivations. □

A.41 Theorem [Preservation (Theorem 4.17)]: 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 cannot have $\emptyset \vdash x : T$.
(T_CONST): $\emptyset \vdash k : \mathsf{ty}(k)$. Contradictory—values do not 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.40), $\emptyset \vdash \sigma(T)$. 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.24).

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

(E_REDUCE/E_BETA): $(\lambda x{:}T_1.\ e_{12})\ v_2 \longrightarrow [v_2/x]e_{12}$. First, we have $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1' \rightarrow T_2')$. By inversion for lambdas (Lemma A.35), $x{:}T_1 \vdash e_{12} : T_2$. Moreover, $x{:}T_1 \rightarrow T_2 \equiv x{:}T_1' \rightarrow T_2'$, which means $T_2 \equiv T_2'$ (Lemma A.21).
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.24), 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.36), $\_{:}\sigma(T) \rightarrow \sigma(T) \equiv x{:}T_1' \rightarrow T_2'$ and $\emptyset \vdash \sigma(T)$. In particular, we have $\sigma(T) \equiv T_2'$ and $\sigma(T) \equiv T_1'$ (Lemma A.21). By substitutivity of conversion (Lemma A.24), $[v_2/x]\sigma(T) \equiv [v_2/x]T_2'$. 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 $\sigma' = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}$. We have $\sigma(T_1) = \sigma'(T_1)$ and $\sigma(T_2) = \sigma'(T_2)$. We restate the typing judgment and its inversion:

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

By cast inversion (Lemma A.36), 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 $\_{:}\sigma(\{x{:}T_1 \mid e\}) \rightarrow \sigma(T_2) \equiv y{:}T_1' \rightarrow T_2'$ and $\{x{:}T_1 \mid e\} \parallel T_2$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$. Inverting this conversion (Lemma A.21), 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 $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'} : y{:}\sigma(T_1) \rightarrow \sigma(T_2)$, 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.24. We are done by T_CONV.
(E_REDUCE/E_PRECHECK):

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

where $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}$. 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:

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

By cast inversion (Lemma A.36), $\emptyset \vdash \sigma(T_1)$ and $\emptyset \vdash \sigma(\{x{:}T_2 \mid e\})$, and $y{:}\sigma(T_1) \rightarrow \sigma(\{x{:}T_2 \mid e\}) \equiv y{:}T_1' \rightarrow T_2'$ 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 $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2} : y{:}\sigma(T_1) \rightarrow \sigma(T_2)$. Note, however, that y does not 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 $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1} : y{:}\sigma(T_2) \rightarrow \sigma(\{x{:}T_2 \mid e\})$ 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 is not 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.24), 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:

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

By cast inversion (Lemma A.36), $\emptyset \vdash \sigma(\{x{:}T \mid e\})$ and $\emptyset \vdash \sigma(T)$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$. Moreover, $y{:}\sigma(T) \rightarrow \sigma(\{x{:}T \mid e\}) \equiv y{:}T_1' \rightarrow T_2'$, where y does not 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.33), we find $\emptyset \vdash [v_2/x]\sigma(e) : \mathsf{Bool}$. Since $[v_2/x]\sigma = \sigma$, by Lemma A.4, $[v_2/x]\sigma(e) = \sigma([v_2/x]e)$. Finally, $\sigma([v_2/x]e) \longrightarrow^* \sigma([v_2/x]e)$ by reflexivity (Lemma A.19).
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.24), $[v_2/y]\sigma(\{x{:}T \mid e\}) \equiv [v_2/y]\,T_2'$. Since y does not 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 $\sigma_i = \sigma|_{\mathrm{AFV}(T_{1i}) \cup \mathrm{AFV}(T_{2i})}$ ($i \in \{1,2\}$). 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:

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

By inversion of this last (Lemma A.21):

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

So by T_CONV and C_SYM, we have $\emptyset \vdash v_2 : \sigma(x{:}T_{11} \rightarrow T_{12})$. By weakening (Lemma A.31), $x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash v_2 : \sigma(x{:}T_{11} \rightarrow T_{12})$.
By inversion of the well formedness of the function types:

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

By weakening (Lemma A.31), we find $x{:}\sigma(T_{21}) \vdash \sigma(T_{11})$ and $x{:}\sigma(T_{21}) \vdash \sigma(T_{21})$. By compatibility:

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

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 (compatibility is symmetric, per Lemma A.27). 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.31) and substitution (Lemma A.33), 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.28:

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

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 is not 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.24), we find $x{:}\sigma(T_{21}) \rightarrow \sigma(T_{22}) \equiv [v_2/y]\,T_2'$. 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:

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

By cast inversion (Lemma A.36):

$$\begin{array}{ll} \emptyset \vdash \sigma(\forall\alpha.\,T_1) & \emptyset \vdash \sigma(\forall\alpha.\,T_2) \\ \forall\alpha.\,T_1 \parallel \forall\alpha.\,T_2 & \mathrm{AFV}(\sigma) \subseteq \emptyset \\ \multicolumn{2}{l}{\_{:}\sigma(\forall\alpha.\,T_1) \rightarrow \sigma(\forall\alpha.\,T_2) \equiv x{:}T_1' \rightarrow T_2'} \end{array}$$

By inversion of this last $\sigma(\forall\alpha.\,T_1) \equiv T_1'$ and $\sigma(\forall\alpha.\,T_2) \equiv T_2'$ (Lemma A.21). 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.32), 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 syntactically different from $\sigma(T_1)$. By inversion of the universal type's well formedness, compatibility, type weakening (Lemma A.32), type substitution (Lemma A.34) 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 does not 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 term substitutivity of conversion (Lemma A.24), $[v_2/x]\forall\alpha.\sigma(T_2) \equiv [v_2/x]\,T_2'$. Since x is not in $\forall\alpha.\sigma(T_2)$, we know that $\forall\alpha.\sigma(T_2) \equiv [v_2/x]\,T_2'$. 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.19) and substitutivity (Lemma A.24).
(E_BLAME): $E\,[\Uparrow l] \longrightarrow \Uparrow l$ $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma A.40), 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 do not 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.37): $\alpha \vdash e' : T''$ and $\forall\alpha.T'' \equiv \forall\alpha.T'$; by inversion of this last (Lemma A.23), $T'' \equiv T'$.
By type variable substitution (Lemma A.34), $\emptyset \vdash [T/\alpha]e' : [T/\alpha]\,T''$. By type substitutivity of conversion (Lemma A.25), $[T/\alpha]\,T'' \equiv [T/\alpha]\,T'$. 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.40), 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 do not 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.40), 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 does not 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 do not step.

(T_FORGET): $\emptyset \vdash v : T$. This case is contradictory—values do not step.
$\square$

A.4. Parametricity

This section proves parametricity; an outline of the proof is described in Section 5.2. We write $R_{T,\theta,\delta}$ for $\{(r_1, r_2) \mid r_1 \sim r_2 : T; \theta; \delta\}$.

A.42 Lemma [Term compositionality (Lemma 5.1)]: If $\theta_1(\delta_1(e)) \longrightarrow^* v_1$ and $\theta_2(\delta_2(e)) \longrightarrow^* v_2$ then $r_1 \sim r_2 : T; \theta; \delta[(v_1, v_2)/x]$ 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 is 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:

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

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 A.16), and we are done. $\square$

A.43 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.42. $\square$

A.44 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.42. $\square$

A.45 Lemma [Type compositionality (Lemma 5.2)]:
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ 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.42, we treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it is 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.
($T = x{:}T_1 \rightarrow T_2$): There are two cases:

($\Rightarrow$): Given $v_1 \sim v_2 : (x{:}T_1 \rightarrow T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : [T'/\alpha](x{:}T_1 \rightarrow T_2); \theta; \delta$. 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.43 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, $v_1\, v_1'' \simeq v_2\, v_2'' : [T'/\alpha]\,T_2; \theta; \delta[(v_1', v_2')/x]$.

($\Leftarrow$): This case is similar: Given $v_1 \sim v_2 : [T'/\alpha](x{:}T_1 \rightarrow T_2); \theta; \delta$, we wish to show that $v_1 \sim v_2 : (x{:}T_1 \rightarrow T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. 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.43 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

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

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

($\Rightarrow$): 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, $v_1 \sim v_2 : \forall\alpha'.([T'/\alpha]\,T_0); \theta; \delta$.

($\Leftarrow$): 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.46 Lemma [Convertibility (Lemma 5.3)]: 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 $r_2 = v_2$.

(C_VAR): It must be that $T_1 = T_2 = \alpha$, so we are done immediately.
(C_BASE): It must be that $T_1 = T_2 = B$, so we are done immediately.
(C_REFINE): 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 A.16):

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

We have $[v_i/x](\theta_i(\delta_i(\sigma_j(e)))) = \sigma_j([v_i/x](\theta_i(\delta_i(e))))$ for $i, j \in \{1, 2\}$ since all substitutions here are closing.
(C_FUN): 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.
(C_FORALL): 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.
(C_SYM): By the IH.
(C_TRANS): By the IHs. □

A.47 Lemma [Cast reflexivity (Lemma 5.4)]: 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.
($T_1 = T_2$): 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

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

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.43 finishes the case.
□

A.48 Theorem [Parametricity (Theorem 5.5)]: (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.
(T_VAR): Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T; \theta; \delta$, which follows from the assumption.
(T_CONST): By the assumption that constants are assigned correct types.
(T_OP): By the assumption that operators are assigned correct types (and the IHs for the operator's arguments).
(T_ABS): 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.

(T_APP): 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 $r_{11} \sim r_{12} : x{:}T_1 \to T_2; \theta; \delta$ and $r_{21} \simeq r_{22} : T_1; \theta; \delta$, respectively. If $r_{11} = r_{12} = \Uparrow l$ or $r_{21} = r_{22} = \Uparrow l$ for some l, then 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.42,

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

By expansion, we can then see

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

(T_TABS): 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 $\Gamma, \alpha \vdash e_0 \simeq e_0 : T_0$.

(T_TAPP): 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.45, $r_1' \sim r_2' : [T_2/\alpha]T_0; \theta; \delta$. 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.47,

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

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.46.
(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

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

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.
□