Breaking Through the Normalization Barrier: A Self-Interpreter for F-omega

Matt Brown
UCLA
msb@cs.ucla.edu

Jens Palsberg
UCLA
palsberg@ucla.edu

Abstract

According to conventional wisdom, a self-interpreter for a strongly normalizing λ-calculus is impossible. We call this the normalization barrier. The normalization barrier stems from a theorem in computability theory that says that a total universal function for the total computable functions is impossible. In this paper we break through the normalization barrier and define a self-interpreter for System F_ω, a strongly normalizing λ-calculus. After a careful analysis of the classical theorem, we show that static type checking in F_ω can exclude the proof's diagonalization gadget, leaving open the possibility for a self-interpreter. Along with the self-interpreter, we program four other operations in F_ω, including a continuation-passing style transformation. Our operations rely on a new approach to program representation that may be useful in theorem provers and compilers.

Categories and Subject Descriptors D.3.4 [*Processors*]: Interpreters; D.2.4 [*Program Verification*]: Correctness proofs, formal methods

General Terms Languages; Theory

Keywords Lambda Calculus; Self Representation; Self Interpretation; Meta Programming

1. Introduction

Barendregt's notion of a *self-interpreter* is a program that recovers a program from its representation and is implemented in the language itself [4]. Specifically for λ-calculus, the challenge is to devise a quoter that maps each term e to a representation $\overline{\mathsf{e}}$, and a self-interpreter u (for *unquote*) such that for every λ-term e we have $(\mathsf{u}\ \overline{\mathsf{e}}) \equiv_\beta \mathsf{e}$. The quoter is an injective function from λ-terms to representations, which are λ-terms in normal form. Barendregt used Church numerals as representations, while in general one can use any β-normal terms as representations. For untyped λ-calculus, in 1936 Kleene presented the first self-interpreter [20], and in 1992 Mogensen presented the first *strong* self-interpreter u that satisfies the property $(\mathsf{u}\ \overline{\mathsf{e}}) \longrightarrow_\beta^* \mathsf{e}$ [23]. In 2009, Rendel, Ostermann, and Hofer [29] presented the first self-interpreter for a *typed* λ-calculus (F_ω^*), and in previous work [8] we presented the first self-interpreter for a typed λ-calculus with *decidable* type checking (Girard's System U). Those results are all for non-normalizing λ-calculi and they go about as far as one can go before reaching what we call the *normalization barrier*.

The normalization barrier: According to conventional wisdom, a self-interpreter for a strongly normalizing λ-calculus is impossible.

The normalization barrier stems from a theorem in computability theory that says that a total universal function for the total computable functions is impossible. Several books, papers, and web pages have concluded that the theorem about total universal functions carries over to self-interpreters for strongly normalizing languages. For example, Turner states that "For any language in which all programs terminate, there are always terminating programs which cannot be written in it - among these are the interpreter for the language itself" [37, pg. 766]. Similarly, Stuart writes that "Total programming languages are still very powerful and capable of expressing many useful computations, but one thing they can't do is interpret themselves" [33, pg. 264]. Additionally, the Wikipedia page on the *Normalization Property* (accessed in May 2015) explains that a self-interpreter for a strongly normalizing λ-calculus is impossible. That Wikipedia page cites three typed λ-calculi, namely simply typed λ-calculus, System F, and the Calculus of Constructions, each of which is a member of Barendregt's cube of typed λ-calculi [5]. We can easily add examples to that list, particularly the other five corners of Barendregt's λ-cube, including F_ω. The normalization barrier implies that a self-interpreter is impossible for every language in the list. In a seminal paper in 1991 Pfenning and Lee [26] considered whether one can define a self-interpreter for System F or F_ω and found that the answer seemed to be "no".

In this paper we take up the challenge presented by the normalization barrier.

The challenge: Can we define a self-interpreter for a strongly normalizing λ-calculus?

Our result: Yes, we present a strong self-interpreter for the strongly normalizing λ-calculus F_ω; the program representation is *deep* and supports a variety of other operations. We also present a much simpler self-interpreter that works for each of System F, F_ω, and F_ω^+; the program representation is *shallow* and supports no other operations.

Figure 1 illustrates how our result relates to other representations of typed λ-calculi with decidable type check-

POPL '16,, January 20-22, 2016, St. Petersburg, FL, USA.
Copyright © 2016 ACM 978-1-4503-3549-2/16/01...\$15.00.
http://dx.doi.org/10.1145/2837614.2837623

$$\begin{tikzcd} \mathrm{F} \arrow[loop above, "\text{[this paper]}"] \arrow[r, "\text{[26]}"] & \mathrm{F}_\omega \arrow[loop above, "\text{[this paper]}"] \arrow[r, "\text{[26]}"] & \mathrm{F}_\omega^+ \arrow[loop above, "\text{[this paper]}"] \arrow[r, "\text{[8]}", "\text{normalization barrier}"'] & \mathrm{U} \arrow[loop above, "\text{[8]}"] \end{tikzcd}$$

Figure 1: Four typed λ-calculi: $\rightarrow$ denotes "represented in."

ing. The normalization barrier separates the three strongly-normalizing languages on the left from System U on the right, which is not strongly-normalizing. Pfenning and Lee represented System F in F_ω, and F_ω in F_ω^+. In previous work we showed that F_ω^+ can be represented in System U, and that System U can represent itself. This paper contributes the self-loops on F, F_ω, and F_ω^+, depicting the first self-representations for strongly-normalizing languages.

Our result breaks through the normalization barrier. The conventional wisdom underlying the normalization barrier makes an implicit assumption that all representations will behave like their counterpart in the computability theorem, and therefore the theorem must apply to them as well. This assumption excludes other notions of representation, about which the theorem says nothing. Thus, our result does not contradict the theorem, but shows that the theorem is less far-reaching than previously thought.

Our result relies on three technical insights. First, we observe that the proof of the classical theorem in computability theory relies on a diagonalization gadget, and that a typed representation can ensure that the gadget fails to type check in F_ω, so the proof doesn't necessarily carry over to F_ω. Second, for our deep representation we use a novel *extensional* approach to representing polymorphic terms. We use instantiation functions that describe the relationship between a quantified type and one of its instance types. Each instantiation function takes as input a term of a quantified type, and instantiates it with a particular parameter type. Third, for our deep representation we use a novel representation of types, which helps us type check a continuation-passing-style transformation.

We present five self-applicable operations on our deep representation, namely a strong self-interpreter, a continuation-passing-style transformation, an intensional predicate for testing whether a closed term is an abstraction or an application, a size measure, and a normal-form checker. Our list of operations extends those of previous work [8].

Our deep self-representation of F_ω could be useful for type-checking self-applicable meta-programs, with potential for applications in typed macro systems, partial evaluators, compilers, and theorem provers. In particular, F_ω is a subset of the proof language of the Coq proof assistant, and Morrisett has called F_ω *the workhorse of modern compilers* [24].

Our deep representation is the most powerful self-representation of F_ω that we have identified: it supports all the five operations listed above. One can define several other representations for F_ω by using fewer of our insights. Ultimately, one can define a *shallow* representation that supports only a self-interpreter and nothing else. As a stepping stone towards explaining our main result, we will show a shallow representation and a self-interpreter in Section 3.3. That representation and self-interpreter have the distinction of working for System F, F_ω and F_ω^+. Thus, we have solved the two challenges left open by Pfenning and Lee [26].

Rest of the paper. In Section 2 we describe F_ω, in Section 3 we analyze the normalization barrier, in Section 4 we describe instantiation functions, in Section 5 we show how to represent types, in Section 6 we show how to represent terms, in Section 7 we present our operations on program representations, in Section 8 we discuss our implementation and experiments, in Section 9 we discuss various aspects of our result, and in Section 10 we compare with related work. Proofs of theorems stated throughout the paper are provided in an appendix that is available from our website [1].

2. System F_ω

System F_ω is a typed λ-calculus within the λ-cube [5]. It combines two axes of the cube: polymorphism and higher-order types (type-level functions). In this section we summarize the key properties of System F_ω used in this paper. We refer readers interested in a complete tutorial to other sources [5, 27]. We give a definition of F_ω in Figure 2. It includes a grammar, rules for type formation and equivalence, and rules for term formation and reduction. The grammar defines the kinds, types, terms, and environments. As usual, types classify terms, kinds classify types, and environments classify free term and type variables. Every syntactically well-formed kind and environment is legal, so we do not include separate formation rules for them. The type formation rules determine the legal types in a given environment, and assigns a kind to each legal type. Similarly, the term formation rules determine the legal terms in a given environment, and assigns a type to each legal term. Our definition is similar to Pierce's [27], with two differences: we use a slightly different syntax, and our semantics is arbitrary β-reduction instead of call-by-value.

It is well known that type checking is decidable, and that types of F_ω terms are unique up to equivalence. We will write $\mathsf{e} \in \mathrm{F}_\omega$ to mean "e is a well-typed term in F_ω". Any well-typed term in System F_ω is strongly normalizing, meaning there is no infinite sequence of β-reductions starting from that term. If we β-reduce enough times, we will eventually reach a term in β-normal form that cannot be reduced further. Formally, term e is β-normal if there is no e' such that $\mathsf{e} \longrightarrow \mathsf{e}'$. We require that representations of terms be *data*, which for λ-calculus usually means a term in β-normal form.

3. The Normalization Barrier

In this section, we explore the similarity of a universal computable function in computability theory and a self-interpreter for a programming language. As we shall see, the exploration has a scary beginning and a happy ending. At first, a classical theorem in computability theory seems to imply that a self-interpreter for F_ω is impossible. Fortunately, further analysis reveals that the proof relies on an assumption that a diagonalization gadget can always be defined for a language with a self-interpreter. We show this assumption to be false: by using a *typed representation*, it is possible to define a self-interpreter such that the diagonalization gadget cannot possibly type check. We conclude the section by demonstrating a simple typed self-representation and a self-interpreter for F_ω.

$$\begin{array}{rl}
\text{(kinds)} & \kappa ::= * \mid \kappa_1 \to \kappa_2 \\
\text{(types)} & \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
\text{(terms)} & \mathtt{e} ::= \mathtt{x} \mid \lambda\mathtt{x}{:}\tau.\mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2 \mid \Lambda\alpha{:}\kappa.\mathtt{e} \mid \mathtt{e}\ \tau \\
\text{(environments)} & \Gamma ::= \langle\rangle \mid \Gamma,(\mathtt{x}{:}\tau) \mid \Gamma,(\alpha{:}\kappa)
\end{array}$$

Grammar

$$\frac{(\alpha{:}\kappa) \in \Gamma}{\Gamma \vdash \alpha : \kappa}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma,(\alpha{:}\kappa) \vdash \tau : *}{\Gamma \vdash (\forall\alpha{:}\kappa.\tau) : *}$$

$$\frac{\Gamma,(\alpha{:}\kappa_1) \vdash \tau : \kappa_2}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau) : \kappa_1 \to \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}$$

Type Formation

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau \equiv \sigma}{\sigma \equiv \tau} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_1 \equiv \sigma_1 \qquad \tau_2 \equiv \sigma_2}{\tau_1 \to \tau_2 \equiv \sigma_1 \to \sigma_2} \qquad \frac{\tau \equiv \sigma}{(\forall\alpha{:}\kappa.\tau) \equiv (\forall\alpha{:}\kappa.\sigma)}$$

$$\frac{\tau \equiv \sigma}{(\lambda\alpha{:}\kappa.\tau) \equiv (\lambda\alpha{:}\kappa.\sigma)} \qquad \frac{\tau_1 \equiv \sigma_1 \qquad \tau_2 \equiv \sigma_2}{\tau_1\ \tau_2 \equiv \sigma_1\ \sigma_2}$$

$$\frac{}{(\lambda\alpha{:}\kappa.\tau) \equiv (\lambda\beta{:}\kappa.\tau[\alpha := \beta])} \qquad \frac{}{(\lambda\alpha{:}\kappa.\tau)\ \sigma \equiv (\tau[\alpha := \sigma])}$$

Type Equivalence

$$\frac{(\mathtt{x}{:}\tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e} : \tau_2}{\Gamma \vdash (\lambda\mathtt{x}{:}\tau_1.\mathtt{e}) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \qquad \Gamma \vdash \mathtt{e}_2 : \tau_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau}$$

$$\frac{\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e} : \tau}{\Gamma \vdash (\Lambda\alpha{:}\kappa.\mathtt{e}) : (\forall\alpha{:}\kappa.\tau)}$$

$$\frac{\Gamma \vdash \mathtt{e} : (\forall\alpha{:}\kappa.\tau) \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash \mathtt{e}\ \sigma : \tau[\alpha{:=}\sigma]}$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathtt{e} : \sigma}$$

Term Formation

$$\begin{array}{c}
(\lambda\mathtt{x}{:}\tau.\mathtt{e})\ \mathtt{e}_1 \longrightarrow \mathtt{e}[\mathtt{x} := \mathtt{e}_1] \\
(\Lambda\alpha{:}\kappa.\mathtt{e})\ \tau \longrightarrow \mathtt{e}[\alpha := \tau]
\end{array}$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_2}{\begin{array}{c}\mathtt{e}_1\ \mathtt{e}_3 \longrightarrow \mathtt{e}_2\ \mathtt{e}_3 \\ \mathtt{e}_3\ \mathtt{e}_1 \longrightarrow \mathtt{e}_3\ \mathtt{e}_2 \\ \mathtt{e}_1\ \tau \longrightarrow \mathtt{e}_2\ \tau \\ (\lambda\mathtt{x}{:}\tau.\mathtt{e}_1) \longrightarrow (\lambda\mathtt{x}{:}\tau.\mathtt{e}_2) \\ (\Lambda\alpha{:}\kappa.\mathtt{e}_1) \longrightarrow (\Lambda\alpha{:}\kappa.\mathtt{e}_2)\end{array}}$$

Reduction

Figure 2: Definition of System F_ω

3.1 Functions from Numbers to Numbers

We recall a classical theorem in computability theory (Theorem 3.2). The proof of the theorem is a diagonalization argument, which we divide into two steps: first we prove a key property (Theorem 3.1) and then we proceed with the proof of Theorem 3.2.

Let $\mathbb{N}$denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Let $\overline{\cdot}$ be an injective function that maps each total, computable function in $\mathbb{N} \to \mathbb{N}$ to an element of $\mathbb{N}$.

We say that $\mathsf{u} \in (\mathbb{N} \times \mathbb{N}) \rightharpoonup \mathbb{N}$ is a universal function for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, if for every total, computable function $\mathtt{f}$ in $\mathbb{N} \to \mathbb{N}$, we have $\forall \mathtt{v} \in \mathbb{N}$: $\mathtt{u(\overline{f},v) = f(v)}$. The symbol $\rightharpoonup$ denotes that $\mathtt{u}$ may be a partial function. Indeed, Theorem 3.2 proves that $\mathtt{u}$ *must* be partial. We let $\mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ denote the set of universal functions for the total, computable functions in $\mathbb{N} \to \mathbb{N}$.

Given a function $\mathtt{u}$ in $(\mathbb{N}\times\mathbb{N}) \rightharpoonup \mathbb{N}$, we define the function $\mathtt{p_u}$ in $\mathbb{N} \to \mathbb{N}$, where $\mathtt{p_u(x) = u(x,x) + 1}$.

Theorem 3.1. *If* $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$*, then* $\mathtt{p_u}$ *isn't total.*

Proof. Suppose $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and $\mathtt{p_u}$ is total. Notice that $\mathtt{p_u}$ is a total, computable function in $\mathbb{N} \to \mathbb{N}$ so $\overline{\mathtt{p_u}}$ is defined. We calculate:

$$\mathtt{p_u(\overline{p_u}) = u(\overline{p_u},\overline{p_u}) + 1 = p_u(\overline{p_u}) + 1}$$

Given that $\mathtt{p_u}$ is total, we have that $\mathtt{p_u(\overline{p_u})}$ is defined; let us call the result v. From $\mathtt{p_u(\overline{p_u}) = p_u(\overline{p_u}) + 1}$, we get $\mathtt{v = v + 1}$, which is impossible. So we have reached a contradiction, hence our assumption (that $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and $\mathtt{p_u}$ is total) is wrong. We conclude that if $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$, then $\mathtt{p_u}$ isn't total. □

Theorem 3.2. *If* $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$*, then* $\mathtt{u}$ *isn't total.*

Proof. Suppose $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and $\mathtt{u}$ is total. For every $\mathtt{x} \in \mathbb{N}$, we have that $\mathtt{p_u(x) = u(x,x) + 1}$. Since $\mathtt{u}$ is total, $\mathtt{u(x,x) + 1}$ is defined, and therefore $\mathtt{p_u(x)}$ is also defined. Since $\mathtt{p_u(x)}$ is defined for every $\mathtt{x} \in \mathbb{N}$, $\mathtt{p_u}$ is total. However, Theorem 3.1 states that $\mathtt{p_u}$ is not total. Thus we have reached a contradiction, so our assumption (that $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and $\mathtt{u}$ is total) is wrong. We conclude that if $\mathtt{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$, then $\mathtt{u}$ isn't total. □

Intuitively, Theorem 3.2 says that if we write an interpreter for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, then that interpreter must go into an infinite loop on some inputs.

3.2 Typed λ-Calculus: $\mathbf{F}_\omega$

Does Theorem 3.2 imply that a self-interpreter for F_ω is impossible? Recall that every well-typed term in F_ω is strongly normalizing. So, if we have a self-interpreter $\mathtt{u}$ for F_ω and we have $\mathtt{(u\ e)} \in F_\omega$, then $\mathtt{(u\ e)}$ is strongly normalizing, which is intuitively expresses that $\mathtt{u}$ is a total function. Thus, Theorem 3.2 seems to imply that a self-interpreter for F_ω is impossible. This is the normalization barrier. Let us examine this intuition via a "translation" of Section 3.1 to F_ω.

Let us recall the definition of a self-interpreter from Section 1, here for F_ω. A quoter is an injective function from terms in F_ω to their representations, which are β-normal terms in F_ω. We write $\overline{\mathsf{e}}$ to denote the representation of a term e. We say that $\mathsf{u} \in F_\omega$ is a self-interpreter for F_ω, if $\forall \mathsf{e} \in F_\omega$: $(\mathsf{u}\ \overline{\mathsf{e}}) \equiv_\beta \mathsf{e}$. We allow $(\mathsf{u}\ \overline{\mathsf{e}})$ to include type abstractions or applications as necessary, and leave them implicit. We use $\mathsf{SelfInt}(F_\omega)$ to denote the set of self-interpreters for F_ω.

Notice a subtle difference between the definition of a universal function in Section 3.1 and the definition of a self-interpreter. The difference is that a universal function takes both its arguments at the same time, while, intuitively, a self-interpreter is *curried* and takes its arguments one by one. This difference plays no role in our further analysis.

Notice also the following consequences of the two requirements of a quoter. The requirement that a quoter must produce terms in β-normal form rules out the identity function as a quoter, because it maps reducible terms to reducible terms. The requirement that a quoter must be injective rules out the function that maps each term to its normal form, because it maps β-equivalent terms to the same β-normal form.

The proof of Theorem 3.1 relies on the diagonalization gadget $(\mathsf{p_u}\ \overline{\mathsf{p_u}})$, where $\mathsf{p_u}$ is a cleverly defined function. The idea of the proof is to achieve the equality $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) = (\mathsf{p_u}\ \overline{\mathsf{p_u}}) + 1$. For the F_ω version of Theorem 3.1, our idea is to achieve the equality $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \equiv_\beta \lambda\mathsf{y}.(\mathsf{p_u}\ \overline{\mathsf{p_u}})$, where y is fresh. Here, $\lambda\mathsf{y}$ plays the role of "+1". Given $\mathsf{u} \in F_\omega$, we define $\mathsf{p_u} = \lambda\mathsf{x}.\ \lambda\mathsf{y}.\ ((\mathsf{u}\ \mathsf{x})\ \mathsf{x})$, where x,y are fresh, and where we omit suitable type annotations for x,y. We can now state an F_ω version of Theorem 3.1.

Theorem 3.3. *If* $\mathsf{u} \in \mathsf{SelfInt}(F_\omega)$*, then* $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \notin F_\omega$*.*

Proof. Suppose $\mathsf{u} \in \mathsf{SelfInt}(F_\omega)$ and $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \in F_\omega$. We calculate:

$$\begin{aligned}&\mathsf{p_u}\ \overline{\mathsf{p_u}}\\ \equiv_\beta\ &\lambda\mathsf{y}.\ ((\mathsf{u}\ \overline{\mathsf{p_u}})\ \overline{\mathsf{p_u}})\\ \equiv_\beta\ &\lambda\mathsf{y}.\ (\mathsf{p_u}\ \overline{\mathsf{p_u}})\end{aligned}$$

From $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \in F_\omega$ we have that $(\mathsf{p_u}\ \overline{\mathsf{p_u}})$ is strongly normalizing. From the Church-Rosser property of F_ω, we have that $(\mathsf{p_u}\ \overline{\mathsf{p_u}})$ has a unique normal form; let us call it v. From $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \equiv_\beta \lambda\mathsf{y}.(\mathsf{p_u}\ \overline{\mathsf{p_u}})$ we get $\mathsf{v} \equiv_\beta \lambda\mathsf{y}.\mathsf{v}$. Notice that v and $\lambda\mathsf{y}.\mathsf{v}$ are *distinct* yet β-equivalent normal forms. Now the Church-Rosser property implies that β-equivalent terms must have the *same* normal form. Thus $\mathsf{v} \equiv_\beta \lambda\mathsf{y}.\mathsf{v}$ implies $\mathsf{v} \equiv_\alpha \lambda\mathsf{y}.\mathsf{v}$, which is false. So we have reached a contradiction, hence our assumption (that $\mathsf{u} \in \mathsf{SelfInt}(F_\omega)$ and $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \in F_\omega$) is wrong. We conclude that if $\mathsf{u} \in \mathsf{SelfInt}(F_\omega)$, then $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \notin F_\omega$. ☐

What is an F_ω version of Theorem 3.2? Given that every term in F_ω is "total" in the sense described earlier, Theorem 3.2 suggests that we should expect $\mathsf{SelfInt}(F_\omega) = \emptyset$. However this turns out to be wrong and indeed in this paper we will define a self-representation and self-interpreter for F_ω. So, $\mathsf{SelfInt}(F_\omega) \neq \emptyset$.

We saw earlier that Theorem 3.1 helped prove Theorem 3.2. Why does Theorem 3.3 fail to lead the conclusion $\mathsf{SelfInt}(F_\omega) = \emptyset$? Observe that in the proof of Theorem 3.2, the key step was to notice that if u is total, also $\mathsf{p_u}$ is total, which contradicts Theorem 3.1. In contrast, the assumption $\mathsf{u} \in \mathsf{SelfInt}(F_\omega)$ produces no useful conclusion like $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \in F_\omega$ that would contradict Theorem 3.3. In particular, it is possible for u and $\mathsf{p_u}$ to be typeable in F_ω, and yet for $(\mathsf{p_u}\ \overline{\mathsf{p_u}})$ to be untypeable. So, the door is open for a self-interpreter for F_ω.

3.3 A Self-Interpreter for F_ω

Inspired by the optimism that emerged in Section 3.2, let us now define a quoter and a self-interpreter for F_ω. The quoter will support *only* the self-interpreter and nothing else. The idea of the quoter is to use a designated variable `id` to block the reduction of every application. The self-interpreter unblocks reduction by substituting the polymorphic identity function for `id`. Below we define the representation $\overline{\mathsf{e}}$ of a closed term e.

$$\Gamma \vdash \mathsf{x} : \tau \rhd \mathsf{x}$$

$$\frac{\Gamma,(\mathsf{x}{:}\tau_1) \vdash \mathsf{e} : \tau_2 \rhd \mathsf{q}}{\Gamma \vdash (\lambda\mathsf{x}{:}\tau_1.\mathsf{e}) : \tau_1 \to \tau_2 \rhd (\lambda\mathsf{x}{:}\tau_1.\mathsf{q})}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \tau_2 \to \tau \rhd \mathsf{q}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \tau_2 \rhd \mathsf{q}_2}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \tau \rhd \mathtt{id}\ (\tau_2 \to \tau)\ \mathsf{q}_1\ \mathsf{q}_2}$$

$$\frac{\Gamma,\alpha{:}\kappa \vdash \mathsf{e} : \tau \rhd \mathsf{q}}{\Gamma \vdash (\Lambda\alpha{:}\kappa.\mathsf{e}) : (\forall\alpha{:}\kappa.\tau) \rhd (\Lambda\alpha{:}\kappa.\mathsf{q})}$$

$$\frac{\Gamma \vdash \mathsf{e} : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathsf{q} \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \mathsf{e}\ \tau_2 : (\tau_1[\alpha := \tau_2]) \rhd \mathtt{id}\ (\forall\alpha{:}\kappa.\tau_1)\ \mathsf{q}\ \tau_2}$$

$$\frac{\Gamma \vdash \mathsf{e} : \tau \rhd \mathsf{q} \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathsf{e} : \sigma \rhd \mathsf{q}}$$

$$\frac{\langle\rangle \vdash \mathsf{e} : \tau \rhd \mathsf{q}}{\overline{\mathsf{e}} = \lambda\mathtt{id}{:}(\forall\alpha{:}*.\ \alpha \to \alpha).\ \mathsf{q}}$$

Our representation is defined in two steps. First, the rules of the form $\Gamma \vdash \mathsf{e} : \tau \rhd \mathsf{q}$ build a pre-representation q from the typing judgment of a term e. The types are needed to instantiate each occurrence of the designated variable `id`. The representation $\overline{\mathsf{e}}$ is defined by abstracting over `id` in the pre-representation. Our self-interpreter takes a representation as input and applies it to the polymorphic identity function:

$$\begin{aligned}\mathtt{unquote} &: \forall\alpha{:}*.\ ((\forall\beta{:}*.\beta \to \beta) \to \alpha) \to \alpha\\ &= \Lambda\alpha{:}*.\ \lambda\mathsf{q}{:}(\forall\beta{:}*.\beta \to \beta) \to \alpha.\\ &\quad\ \mathsf{q}\ (\Lambda\beta{:}*.\ \lambda\mathsf{x}{:}\beta.\ \mathsf{x})\end{aligned}$$

Theorem 3.4.
If $\langle\rangle \vdash \mathsf{e} : \tau$*, then* $\langle\rangle \vdash \overline{\mathsf{e}} : (\forall\alpha{:}*.\ \alpha \to \alpha) \to \tau$ *and* $\overline{\mathsf{e}}$ *is in* β*-normal form.*

Theorem 3.5.
If $\langle\rangle \vdash \mathsf{e} : \tau$*, then* $\mathtt{unquote}\ \tau\ \overline{\mathsf{e}} \longrightarrow^* \mathsf{e}$.

This self-interpreter demonstrates that it is possible to break through the normalization barrier. In fact, we can define a similar self-representation and self-interpreter for System F and for System F_ω^+. However, the representation supports no other operations than `unquote`: parametricity implies that the polymorphic identity function is the *only* possible argument to a representation $\overline{\mathsf{e}}$ [38]. The situation is similar to the one faced by Pfenning and Lee who observed that "evaluation is just about the only useful function definable" for their representation of F_ω in F_ω^+. We call a representation *shallow* if it supports only one operation, and we call representation *deep* if it supports a variety of operations. While the representation above is shallow, we have found it to be a good starting point for designing deep representations.

In Figure 3 we define a *deep* self-representation of F_ω that supports multiple operations, including a self-interpreter, a CPS-transformation, and a normal-form checker. The keys to why this works are two novel techniques along with typed Higher-Order Abstract Syntax (HOAS), all of which we will explain in the following sections. First, in Section 4 we present an extensional approach to representing polymorphism in F_ω. Second, in Section 5 we present a simple representation of types that is sufficient to support our CPS transformation. Third, in Section 6 we present a typed HOAS representation based on Church encoding, which supports operations that fold over the term. Finally, in Section 7 we define five operations for our representation.

4. Representing Polymorphism

In this section, we discuss our extensional approach to representing polymorphic terms in our type Higher-Order Abstract Syntax representation. Our approach allows us to define our HOAS representation of F_ω in F_ω itself. Before presenting our extensional approach, we will review the intensional approach used by previous work. As a running example, we consider how to program an important piece of a self-interpreter for a HOAS representation.

Our HOAS representation, like those of Pfenning and Lee [26], Rendel et al. [29], and our own previous work [8], is based on a typed Church encoding. Operations are defined by cases functions, one for each of λ-abstraction, application, type abstraction, and type application. Our representation differs from the previous work in how we type check the case functions for type abstraction and type applications. Our running example will focus just on the case function for type applications. To simplify further, we consider only the case function for type applications in a self-interpreter.

4.1 The Intensional Approach

The approach of previous work [8, 26, 29] can be demonstrated by a *polymorphic type-application function*, which can apply any polymorphic term to any type in its domain. The function $\texttt{tapp}^+$ defined below is a polymorphic type application function for System F_ω^+. System F_ω^+ extends F_ω with kind abstractions and applications in terms (written $\Lambda\kappa.\texttt{e}$ and $\texttt{e}\ \kappa$ respectively), and kind-polymorphic types (written $\forall^+\kappa.\tau$):

$$\texttt{tapp}^+ : (\forall^+\kappa.\forall\beta{:}\kappa\to*.(\forall\alpha{:}\kappa.\ \beta\ \alpha) \to \forall\gamma{:}\kappa.\beta\ \gamma)$$
$$\texttt{tapp}^+ = \Lambda^+\kappa.\Lambda\beta{:}\kappa\to*.\lambda\texttt{e}{:}(\forall\alpha{:}\kappa.\beta\ \alpha).\ \Lambda\gamma{:}\kappa.\texttt{e}\ \gamma$$

The variables κ and β respectively range over the domain and codomain of an arbitrary quantified type. The domain of $(\forall\alpha_1{:}\kappa_1.\tau_1)$ is the kind κ_1, and the codomain is the type function $(\lambda\alpha_1{:}\kappa_1.\tau_1)$ since τ_1 depends on a type parameter α_1 of kind κ_1. Since the body of a quantified type must have kind $*$, the codomain function $(\lambda\alpha_1{:}\kappa_1.\tau_1)$ must have kind $(\kappa_1 \to *)$. A quantified type can be expressed in terms of its domain and codomain: $(\forall\alpha_1{:}\kappa_1.\tau_1) \equiv (\forall\alpha{:}\kappa_1.\ (\lambda\alpha_1{:}\kappa_1.\tau_1)\ \alpha)$. Similarly, any instance of the quantified type can be expressed as an application of the codomain to a parameter: $(\tau_1[\alpha_1{:=}\tau_2]) \equiv (\lambda\alpha_1{:}\kappa_1.\tau_1)\ \tau_2$. We use these equivalences in the type of $\texttt{tapp}^+$: the quantified type $(\forall\alpha{:}\kappa.\ \beta\ \alpha)$ is expressed in terms of an arbitrary domain κ and codomain β, and the instantiation $\beta\ \gamma$ is expressed as an application of the codomain β to an arbitrary parameter γ.

We call this encoding *intensional* because it encodes the structure of a quantified type by abstracting over its parts (its domain κ and codomain β). This ensures that e can only have a quantified type, and that γ ranges over exactly the types to which e can be applied. In other words, γ can be instantiated with τ_2 if and only if $\texttt{e}\ \tau_2$ is well-typed.

Consider a type application $\texttt{e}\ \tau_2$ with the derivation:

$$\frac{\Gamma \vdash \texttt{e} : (\forall\alpha_1{:}\kappa_1.\tau_1) \qquad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \texttt{e}\ \tau_2 : \tau_1[\alpha_1{:=}\tau_2]}$$

We can encode $\texttt{e}\ \tau_2$ in F_ω^+ as $\texttt{tapp}^+\kappa_1\ (\lambda\alpha_1{:}\kappa_1.\tau_1)\ \texttt{e}\ \tau_2$. However, F_ω does not support kind polymorphism, so $\texttt{tapp}^+$ is not definable in F_ω. To represent F_ω in itself, we need a new approach.

4.2 An Extensional Approach

The approach we use in this paper is *extensional:* rather than encoding the structure of a quantified type, we encode the relationship between a quantified type and its instances. We encode the relationship "$(\tau_1[\alpha{:=}\tau_2])$ is an instance of $(\forall\alpha_1{:}\kappa_1.\tau_1)$" with an *instantiation function* of type $(\forall\alpha_1{:}\kappa_1.\tau_1) \to (\tau_1[\alpha_1{:=}\tau_2])$. An example of such an instantiation function is $\lambda\texttt{x}{:}(\forall\alpha_1{:}\kappa_1.\tau_1).\ \texttt{x}\ \tau_2$ that instantiates an input term of type $(\forall\alpha_1{:}\kappa_1.\tau_1)$ with the type τ_2. For convenience, we define an abbreviation $\texttt{inst}_{(\tau,\sigma)} = \lambda\texttt{x}{:}\tau.\ \texttt{x}\ \sigma$, which is well-typed only when τ is a quantified type and σ is in the domain of τ.

The advantage of using instantiation functions is that all quantified types and all instantiations of quantified types are types of kind $*$. Thus, we can encode the rule for type applications in F_ω by abstracting over the quantified type, the instance type, and the instantiation function for them:

$$\texttt{tapp} : (\forall\alpha{:}*.\ \alpha \to \forall\beta{:}*.\ (\alpha \to \beta) \to \beta)$$
$$\texttt{tapp} = \Lambda\alpha{:}*.\ \lambda\texttt{e}{:}\alpha.\ \Lambda\beta{:}*.\ \lambda\texttt{inst}{:}\alpha \to \beta.\ \texttt{inst e}$$

Using tapp we can encode the type application $\texttt{e}\ \tau_2$ above as

$$\texttt{tapp}\ (\forall\alpha_1{:}\kappa_1.\tau_1)\ \texttt{e}\ (\tau_1[\alpha_1{:=}\tau_2])\ \texttt{inst}_{((\forall\alpha_1{:}\kappa_1.\tau_1),\tau_2)}$$

Unlike the intensional approach, the extensional approach provides no guarantee that e will always have a quantified type. Furthermore, even if e does have a quantified type, inst is not guaranteed to actually be an instantiation function. In short, the intensional approach provides two Free Theorems [38] that we don't get with our extensional approach. However, the extensional approach has the key advantage of enabling a representation of F_ω in itself.

5. Representing Types

We use type representations to type check term representations and operations on term representations. Our type representation is shown as part of the representation of F_ω in Figure 3. The $[\![\tau]\!]$ syntax denotes the pre-representation of the type τ, and $\overline{\tau}$ denotes the representation. A pre-representation is defined using a designated variable F, and a representation abstracts over F.

Our type representation is novel and designed to support three important properties: first, it can represent all types (not just types of kind $*$); second, representation preserves equivalence between types; third, it is expressive enough to typecheck all our benchmark operations. The first and second properties and play an important part in our representation of polymorphic terms.

$$\begin{aligned}[\![\alpha]\!] &= \alpha\\ [\![\tau_1 \to \tau_2]\!] &= \mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]\\ [\![\forall\alpha{:}\kappa.\tau]\!] &= \forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau]\!]\\ [\![\lambda\alpha{:}\kappa.\tau]\!] &= \lambda\alpha{:}\kappa.\ [\![\tau]\!]\\ [\![\tau_1\ \tau_2]\!] &= [\![\tau_1]\!]\ [\![\tau_2]\!]\end{aligned}$$

Pre-Representation of Types

$$\overline{\tau} = \lambda\mathtt{F}{:}* \to *.\ [\![\tau]\!]$$

Representation of Types

```
U  = (∗ → ∗) → ∗
Op = λF:∗ → ∗. λα:U. F (α F)

Strip = λF:∗ → ∗. λα:∗.
          ∀β:*.(∀γ:*. F γ → β) → α → β

Abs  = λF:∗→∗.
         ∀α:∗.∀β:∗.(F α → F β) → F (F α → F β)
App  = λF:∗→∗.
         ∀α:∗.∀β:∗.F (F α → F β) → F α → F β
TAbs = λF:∗→∗.
         ∀α:∗. Strip F α → α → F α
TApp = λF:∗→∗.
         ∀α:∗.F α → ∀β:∗.(α → F β) → F β

Exp = λα:U. ∀F:∗ → ∗.
      Abs F → App F → TAbs F → TApp F →
      Op F α
```

Kind and Type Definitions

$$\mathtt{inst}_{(\tau,\sigma)} = \lambda\mathtt{x}{:}\tau.\ \mathtt{x}\ \sigma$$

Instantiation Functions

$$\frac{}{* \blacktriangleright (\forall\alpha{:}*.\alpha)} \qquad \frac{\kappa \blacktriangleright \sigma}{\kappa_1 \to \kappa \blacktriangleright \lambda\alpha{:}\kappa_1.\sigma}$$

Kind Inhabitants

$$\frac{\kappa \blacktriangleright \sigma}{\begin{aligned}\mathtt{strip}_{(\mathtt{F},\kappa,\tau)} = {}&\Lambda\alpha{:}*.\ \lambda\mathtt{f}{:}(\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha).\\ &\lambda\mathtt{x}{:}(\forall\gamma{:}\kappa.\mathtt{F}\ (\tau\ \gamma)).\ \mathtt{f}\ (\tau\ \sigma)\ (\mathtt{x}\ \sigma)\end{aligned}}$$

Strip Functions

$$\frac{(\mathtt{x}{:}\tau) \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau \rhd \mathtt{x}}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e} : \tau_2 \rhd \mathtt{q}}{\Gamma \vdash (\lambda\mathtt{x}{:}\tau_1.\mathtt{e}) : \tau_1 \to \tau_2 \rhd \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\mathtt{q})}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau \rhd \mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \mathtt{e} : \tau \rhd \mathtt{q}}{\begin{aligned}\Gamma \vdash (\Lambda\alpha{:}\kappa.\mathtt{e}) : (\forall\alpha{:}\kappa.\tau) \rhd \mathtt{tabs}\ &[\![\forall\alpha{:}\kappa.\tau]\!]\\ &\mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])}\\ &(\Lambda\alpha{:}\kappa.\mathtt{q})\end{aligned}}$$

$$\frac{\Gamma \vdash \mathtt{e} : (\forall\alpha{:}\kappa.\tau) \rhd \mathtt{q} \qquad \Gamma \vdash \sigma : \kappa}{\begin{aligned}\Gamma \vdash \mathtt{e}\ \sigma : \tau[\alpha{:=}\sigma] \rhd \mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau]\!]\ \mathtt{q}\ &[\![\tau[\alpha{:=}\sigma]]\!]\\ &\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau]\!],[\![\sigma]\!])}\end{aligned}}$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q} \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathtt{e} : \sigma \rhd \mathtt{q}}$$

Pre-Representation of Terms

$$\frac{\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}}{\begin{aligned}\overline{\mathtt{e}} = {}&\Lambda\mathtt{F}{:}* \to *.\\ &\lambda\mathtt{abs}\ {:}\mathtt{Abs}\ \mathtt{F}.\ \ \lambda\mathtt{app}\ {:}\mathtt{App}\ \mathtt{F}.\\ &\lambda\mathtt{tabs}{:}\mathtt{TAbs}\ \mathtt{F}.\ \lambda\mathtt{tapp}{:}\mathtt{TApp}\ \mathtt{F}.\\ &\mathtt{q}\end{aligned}}$$

Representation of Terms

Figure 3: Representation of F_ω

Type representations support operations that iterate a type function R over the first-order types – arrows and universal quantifiers. Each operation on representations produces results of the form $\mathtt{R}\ ([\![\tau]\!][\mathtt{F} := \mathtt{R}])$, which we call the "interpretation of τ under R". For example, the interpretation of $(\forall\alpha{:}*.\ \alpha \to \alpha)$ under R is $\mathtt{R}\ ([\![\forall\alpha{:}*.\ \alpha \to \alpha]\!][\mathtt{F} := \mathtt{R}]) = \mathtt{R}\ (\forall\alpha{:}*.\ \mathtt{R}\ (\mathtt{R}\ \alpha \to \mathtt{R}\ \alpha))$.

As stated previously, type representations are used to typecheck representations of terms and their operations. In particular, a term of type τ is represented by a term of type $\mathtt{Exp}\ \overline{\tau}$, and each operation on term representation produces results with types that are interpretations under some R.

Let's consider the outputs produced by `unquote`, `size`, and `cps`, when applied to a representation of the polymorphic identity function, which has the type $(\forall\alpha{:}*.\ \alpha \to \alpha)$. For `unquote`, the type function R is the identity function $\mathtt{Id} = (\lambda\alpha{:}*.\alpha)$. Therefore, `unquote` applied to the representation of the polymorphic identity function will produce an output with the type $\mathtt{Id}\ (\forall\alpha{:}*.\ \mathtt{Id}\ (\mathtt{Id}\ \alpha \to \mathtt{Id}\ \alpha)) \equiv (\forall\alpha{:}*.\alpha \to \alpha)$. For `size`, R is the constant function $\mathtt{KNat} = (\lambda\alpha{:}*.\mathtt{Nat})$. Therefore, `size` applied to the representation of the polymorphic identity function will produce an output with the type $\mathtt{KNat}\ (\forall\alpha{:}*.\ \mathtt{KNat}\ (\mathtt{KNat}\ \alpha \to \mathtt{KNat}\ \alpha)) \equiv \mathtt{Nat}$. For `cps`, R is the function $\mathtt{Ct} = (\lambda\alpha{:}*.\ \forall\beta{:}*.\ (\alpha \to \beta) \to \beta)$, such that $\mathtt{Ct}\ \alpha$ is the type of a continuation for values of type α. Therefore, `cps` applied to the representation of the polymorphic identity function will produce an output with the type $\mathtt{Ct}\ (\forall\alpha{:}*.\ \mathtt{Ct}\ (\mathtt{Ct}\ \alpha \to \mathtt{Ct}\ \alpha))$. This type suggests that every sub-term has been transformed into continuation-passing style.

We also represent higher-order types, since arrows and quantifiers can occur within them. Type variables, abstractions, and applications are represented meta-circularly. Intuitively, the pre-representation of an abstraction is an abstraction over pre-representations. Since pre-representations of κ-types (i.e. types of kind κ) are themselves κ-types, an abstraction over κ-types can also abstract over pre-representations of κ-types. In other words, abstractions are represented as themselves. The story is the same for type variables and applications.

Examples. The representation of $(\forall\alpha{:}*.\ \alpha \to \alpha)$ is:

$$\begin{aligned}&\overline{\forall\alpha{:}*.\ \alpha \to \alpha}\\ &= \lambda\mathtt{F}{:}*{\to}*.\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]\\ &= \lambda\mathtt{F}{:}*{\to}*.\ \forall\alpha{:}*.\ \mathtt{F}\ (\mathtt{F}\ \alpha \to \mathtt{F}\ \alpha)\end{aligned}$$

Our representation is defined so that the representations of two β-equivalent types are also β-equivalent. In other words, representation of types preserves β-equivalence. In particular, we can normalize a type before or after representation, with the same result. For example,

$$\begin{aligned}
& \overline{\forall\alpha{:}{*}.(\lambda\gamma{:}{*}.\gamma \to \gamma)\ \alpha} \\
= \ & \lambda\mathsf{F}{:}{*}{\to}{*}.\ [\![\forall\alpha{:}{*}.(\lambda\gamma{:}{*}.\gamma \to \gamma)\ \alpha]\!] \\
= \ & \lambda\mathsf{F}{:}{*}{\to}{*}.\ \forall\alpha{:}{*}.\ \mathsf{F}\ ((\lambda\gamma{:}{*}.\mathsf{F}\ \gamma \to \mathsf{F}\ \gamma)\ \alpha) \\
\equiv_\beta & \lambda\mathsf{F}{:}{*}{\to}{*}.\ \forall\alpha{:}{*}.\ \mathsf{F}\ (\mathsf{F}\ \alpha \to \mathsf{F}\ \alpha) \\
= \ & \overline{\forall\alpha{:}{*}.\ \alpha \to \alpha}
\end{aligned}$$

Properties. We now discuss some properties of our type representation that are important for representing terms. First, we can pre-represent legal types of any kind and in any environment. Since a representation abstracts over the designated type variable F in a pre-representation, the representation of a κ-type is a type of kind $(* \to *) \to \kappa$. In particular, base types (i.e. types of kind $*$) are represented by a type of kind $(* \to *) \to *$. This kind will be important for representing terms, so in Figure 3 we define $\mathsf{U} = (* \to *) \to *$.

Theorem 5.1. *If $\Gamma \vdash \tau : \kappa$, then $\Gamma \vdash \overline{\tau} : (* \to *) \to \kappa$.*

Equivalence preservation relies on the following substitution theorem, which will also be important for our representation of terms.

Theorem 5.2. *For any types τ and σ, and any type variable α, we have $[\![\tau]\!][\alpha := [\![\sigma]\!]] = [\![\tau[\alpha := \sigma]]\!]$.*

We now formally state the equivalence preservation property of type pre-representation and representation.

Theorem 5.3. *$\tau \equiv \sigma$ if and only if $\overline{\tau} \equiv \overline{\sigma}$.*

6. Representing Terms

In this section we describe our representation of F_ω terms. Our representations are typed to ensure that only well-typed terms can be represented. We typecheck representations of terms using type representations. In particular, a term of type τ is represented by a term of type $\mathsf{Exp}\ \overline{\tau}$.

We use a typed Higher-Order Abstract Syntax (HOAS) representation based on Church encodings, similar to those used in previous work [8, 26, 29]. As usual in Higher-Order Abstract Syntax (HOAS), we represent variables and abstractions meta-circularly, that is, using variables and abstractions. This avoids the need to implement capture-avoiding substitution on our operations – we inherit it from the host language implementation. As in our previous work [8], our representation is also parametric (PHOAS) [14, 39]. In PHOAS representations, the types of variables are parametric. In our case, they are parametric in the type function F that defines an interpretation of types.

Our representation of F_ω terms is shown in Figure 3. We define our representation in two steps, as we did for types. The pre-representation of a term is defined using the designated variables F, abs, app, tabs, and tapp. The representation abstracts over these variables in the pre-representation.

While the pre-representation of types can be defined by the type alone, the pre-representation of a term depends on its typing judgment. We call the function that maps typing judgments to pre-representations the *pre-quoter*. We write $\Gamma \vdash \mathsf{e} : \tau \rhd \mathsf{q}$ to denote “given an input judgment $\Gamma \vdash \mathsf{e} : \tau$ the pre-quoter outputs a pre-representation q”. The pre-representation of a term is defined by a type function F that defines pre-representations of types, and by four *case functions* that together define a fold over the structure of a term. The types of each case function depends on the type function F. The case functions are named abs, app, tabs, and tapp, and respectively represent λ-abstraction, function application, type-abstraction, and type application.

The representation $\overline{\mathsf{e}}$ of a closed term e is obtained by abstracting over the variables F, abs, app, tabs, and tapp in the pre-representation of e. If e has type τ, its pre-representation has type $\mathsf{F}\ [\![\tau]\!]$, and its representation has type $\mathsf{Exp}\ \overline{\tau}$. The choice of τ can be arbitrary because typings are unique up to β-equivalence and type representation preserves β-equivalence.

Stripping redundant quantifiers. In addition to the inst functions discussed in Section 4, our quoter embeds a specialized variant of instantiation functions into representations. These functions can strip redundant quantifiers, which would otherwise limit the expressiveness of our HOAS representation. For example, our size operation will use them to remove the redundant quantifier from intermediate values with types of the form $(\forall\alpha{:}\kappa.\mathsf{Nat})$. The type Nat is closed, so α does not occur free in Nat. This is why the quantifier is said to be redundant. This problem of redundant quantifiers is well known, and applies to other HOAS representations as well [29].

We can strip a redundant quantifier with a type application: if e has type $(\forall\alpha{:}\kappa.\mathsf{Nat})$ and σ is a type of kind κ, then $\mathsf{e}\ \sigma$ has the type Nat. We can also use the instantiation function $\mathsf{inst}_{(\forall\alpha:\kappa.\mathsf{Nat}),\sigma}$, which has type $(\forall\alpha{:}\kappa.\mathsf{Nat}) \to \mathsf{Nat}$. The choice of σ is arbitrary – it can be any type of kind κ. It happens that in F_ω all kinds are inhabited, so we can always find an appropriate σ to strip a redundant quantifier.

Our quoter generates a single strip function for each type abstraction in a term and embeds it into the representation. At the time of quotation most quantifiers are not redundant – redundant quantifiers are introduced by certain operations like size. Whether a quantifier will become redundant depends on the result type function F for an operation. In our operations, redundant quantifiers are introduced when F is a constant function. The operation size has results typed using the constant Nat function $\mathsf{KNat} = (\lambda\alpha{:}{*}.\mathsf{Nat})$. Each strip function is general enough to work for multiple operations that introduce redundant quantifiers, and to still allow operations like unquote that need the quantifier.

To provide this generality, the strip functions take some additional inputs that help establish that a quantifier is redundant before stripping it. Each strip function will have a type of the form $\mathsf{Strip}\ \mathsf{F}\ [\![\forall\alpha{:}\kappa.\tau]\!] \equiv (\forall\beta{:}{*}.\ (\forall\gamma{:}{*}.\ \mathsf{F}\ \gamma \to \beta) \to [\![\forall\alpha{:}\kappa.\tau]\!] \to \beta)$. The type F is the type function defines an interpretation of types. The type $[\![\forall\alpha{:}\kappa.\tau]\!]$ is the quantified type with the redundant quantifier to be stripped. Recall that $[\![\forall\alpha{:}\kappa.\tau]\!] = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau]\!])$. The type term of type $(\forall\gamma{:}{*}.\ \mathsf{F}\ \gamma \to \beta)$ shows that F is a constant function that always returns β. The strip function uses it to turn the type $(\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau]\!])$ into the type $(\forall\alpha{:}\kappa.\beta)$ where α has become redundant. For size, we will have $\mathsf{F} = \mathsf{KNat} = (\lambda\alpha{:}{*}.\mathsf{Nat})$. We show that KNat is the constant Nat function with an identity function $(\Lambda\gamma{:}{*}.\lambda\mathsf{x}{:}\mathsf{KNat}\ \gamma.\ \mathsf{x})$. The type of this function is $(\forall\gamma{:}{*}.\mathsf{KNat}\ \gamma \to \mathsf{KNat}\ \gamma)$, which is equivalent to $(\forall\gamma{:}{*}.\mathsf{KNat}\ \gamma \to \mathsf{Nat})$.

Types of case functions. The types of the four case functions abs, app, tabs, and tapp that define an interpretation, respectively Abs, App, TAbs, and TApp, are shown in Figure 3. The types of each function rely on invariants about pre-representations of types. For example, the type App F uses the fact that the pre-representation of an arrow type $[\![\tau_1 \to \tau_2]\!]$ is equal to $\mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$. In other words, `App F` abstracts over the types $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ that can change, and makes explicit the structure $\mathtt{F}\ \alpha \to \mathtt{F}\ \beta$ that is invariant. These types allow the implementation of each case function to use this structure – it is part of the "interface" of representations, and plays an important role in the implementation of each operation.

Building representations. The first rule of pre-representation handles variables. As in our type representation, variables are represented meta-circularly, that is, by other variables. We will re-use the variable name, but change its type: a variable of type τ is represented by a variable of type $\mathtt{F}\ [\![\tau]\!]$. This type is the same as the type of a pre-representation. In other words, variables in a pre-representation range over pre-representations.

The second rule of pre-representation handles λ-abstractions. We recursively pre-quote the body, in which a variable `x` can occur free. Since variables are represented meta-circularly, `x` can occur free in the pre-representation `q` of the body. Therefore, we bind `x` in the pre-representation. This is standard for Higher-Order Abstract Syntax representations. Again, we change of the type of `x` from τ_1 to $\mathtt{F}\ [\![\tau_1]\!]$. It may be helpful to think of `q` as the "open pre-representation of `e`", in the sense that `x` can occur free, and to think of $(\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q})$ as the "closed pre-representation of `e`". The open pre-representation of `e` has type $\mathtt{F}\ [\![\tau_2]\!]$ in an environment that assigns `x` the type $\mathtt{F}\ [\![\tau_1]\!]$. The closed pre-representation of `e` has type $\mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$. The pre-representation of $(\lambda\mathtt{x}{:}\tau_1.\ \mathtt{e})$ is built by applying the case function `abs` to the types $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ and the closed pre-representation of `e`.

The third rule of pre-representation handles applications. We build the pre-representation of an application $\mathtt{e}_1\ \mathtt{e}_2$ by applying the case function `app` to the types $[\![\tau_2]\!]$ and $[\![\tau]\!]$ and the pre-representations of $\mathtt{e}_1$ and $\mathtt{e}_2$.

The fourth rule of pre-representation handles type abstractions. As for λ-abstractions, we call `q` the open pre-representation of `e`, and abstract over α to get the closed pre-representation of `e`. Unlike for λ-abstractions, we do not pass the domain and codomain of the type to the case function `tabs`, since that would require kind-polymorphism as discussed in Section 4. Instead, we pass to `tabs` the pre-representation of the quantified type directly. We also pass to `tabs` a quantifier stripping function that enables `tabs` to remove the quantifier from $[\![\forall\alpha{:}\kappa.\ \mathtt{F}\ \tau]\!]$ in case `F` is a constant function. Note that the strip function is always defined, since $[\![\forall\alpha{:}\kappa.\ \mathtt{F}\ \tau]\!] = \forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!]$.

The fifth rule of pre-quotation handles type applications. We build the pre-representation of a type application $\mathtt{e}\ \sigma$ by applying the case function `tapp` to the pre-representation of the quantified type $[\![\forall\alpha{:}\kappa.\tau]\!]$, the pre-representation of the term `e`, the pre-representation of the instantiation type $[\![\tau[\alpha{:=}\sigma]]\!]$, and the instantiation function $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau]\!],[\![\sigma]\!])}$, which can apply any term of type $[\![\forall\alpha{:}\kappa.\tau]\!]$ to the type $[\![\sigma]\!]$. Since $[\![\forall\alpha{:}\kappa.\tau]\!] = (\forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!])$, the instantiation function has type $[\![\forall\alpha{:}\kappa.\tau]\!] \to \mathtt{F}\ [\![\tau[\alpha{:=}\sigma]]\!]$.

The last rule of pre-quotation handles the type-conversion rule. Unsurprisingly, the pre-representation of `e` is the same when `e` has type σ as when it has type τ. When `e` has type τ, its pre-representation will have type $\mathtt{F}\ [\![\tau]\!]$. When `e` has type σ, its pre-representation will have type $\mathtt{F}\ [\![\sigma]\!]$. By Theorem 5.3, these two types are equivalent, so `q` can be given either type.

Examples. We now give two example representations. Our first example is the representation of the polymorphic identity function $\Lambda\alpha{:}{*}.\lambda\mathtt{x}{:}\alpha.\mathtt{x}$:

$$\begin{array}{l}
\Lambda\mathtt{F}{:}{*} \to {*}.\\
\lambda\mathtt{abs}{:}\mathtt{Abs\ F}.\ \lambda\mathtt{app}{:}\mathtt{App\ F}.\\
\lambda\mathtt{tabs}{:}\mathtt{TAbs\ F}.\ \lambda\mathtt{tapp}{:}\mathtt{TApp\ F}.\\
\mathtt{tabs}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]\ \mathtt{strip}_{(\mathtt{F},{*},[\![\lambda\alpha:*.\alpha\to\alpha]\!])}\\
\quad (\Lambda\alpha{:}{*}.\ \mathtt{abs}\ \alpha\ \alpha\ (\lambda\mathtt{x}{:}\mathtt{F}\ \alpha.\ \mathtt{x}))
\end{array}$$

We begin by abstracting over the type function `F` that defines an interpretation of types, and the four case functions that define an interpretation of terms. Then we build the pre-representation of $\Lambda\alpha{:}{*}.\lambda\mathtt{x}{:}\alpha.\mathtt{x}$. We represent the type abstraction using `tabs`, the term abstraction using `abs`, and the variable `x` as another variable also named `x`.

Our second example is representation of $(\lambda\mathtt{x}{:}(\forall\alpha{:}{*}.\ \alpha \to \alpha).\ \mathtt{x}\ (\forall\alpha{:}{*}.\ \alpha \to \alpha)\ \mathtt{x})$, which applies an input term to itself.

$$\begin{array}{l}
\Lambda\mathtt{F}{:}{*} \to {*}.\\
\lambda\mathtt{abs}{:}\mathtt{Abs\ F}.\ \lambda\mathtt{app}{:}\mathtt{App\ F}.\\
\lambda\mathtt{tabs}{:}\mathtt{TAbs\ F}.\ \lambda\mathtt{tapp}{:}\mathtt{TApp\ F}.\\
\mathtt{abs}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]\\
\ \ (\lambda\mathtt{x}{:}\ \mathtt{F}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!].\\
\ \ \ \mathtt{app}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]\\
\quad\ (\mathtt{tapp}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]\ \mathtt{x}\\
\qquad\quad [\![(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)]\!]\\
\qquad\quad \mathtt{inst}_{([\![\forall\alpha:*.\alpha\to\alpha]\!],[\![\forall\alpha:*.\alpha\to\alpha]\!])})\\
\quad\ \mathtt{x})
\end{array}$$

The overall structure is similar to above: we begin with the five abstractions that define interpretations of types and terms. We then use the case functions to build the pre-representation of the term. The instantiation function $\mathtt{inst}_{([\![\forall\alpha:*.\alpha\to\alpha]\!],[\![\forall\alpha:*.\alpha\to\alpha]\!])}$ has the type $[\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!] \to \mathtt{F}\ [\![(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)]\!]$. Here, the quantified type being instantiated is $[\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!] = \forall\alpha{:}{*}.\ \mathtt{F}\ [\![\alpha \to \alpha]\!]$, the instantiation parameter is also $[\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]$, and the instantiation type is $\mathtt{F}\ [\![(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)]\!]$. By lemma 5.2, we have:

$$\begin{array}{rl}
& (\mathtt{F}\ [\![\alpha \to \alpha]\!])[\alpha := [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]]\\
= & \mathtt{F}\ ([\![\alpha \to \alpha]\!][\alpha := [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]])\\
= & \mathtt{F}\ [\![(\alpha \to \alpha)[\alpha := \forall\alpha{:}{*}.\ \alpha \to \alpha]]\!]\\
= & \mathtt{F}\ [\![(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)]\!]
\end{array}$$

Properties. We typecheck pre-quotations under a modified environment that changes the types of term variables and binds the variables `F`, `abs`, `app`, `tabs`, and `tapp`. The bindings of type variables are unchanged.

The environment for pre-quotations of closed terms only contains bindings for `F`, `abs`, `app`, `tabs`, and `tapp`. The representation of a closed term abstracts over these variables, and so can be typed under an empty environment.

Theorem 6.1. *If* $\langle\rangle \vdash \mathtt{e} : \tau$, *then* $\langle\rangle \vdash \overline{\mathtt{e}} : \mathtt{Exp}\ \overline{\tau}$.

Our representations are *data*, which for F_ω means a β-normal form.

Theorem 6.2. *If* $\langle\rangle \vdash \mathtt{e} : \tau$, *then* $\overline{\mathtt{e}}$ *is* β*-normal.*

Our quoter preserves equality of terms up to equivalence of types. That is, if two terms are equal up to equivalence of types, then their representations are equal up to equivalence of types as well. Our quoter is also injective up to equivalence of types, so the converse is also true: if the representations of two terms are equal up to equivalence of types, then the terms are themselves equal up to equivalence of types.

Definition 6.1 (Equality up to equivalence of types)**.** *We write $\mathtt{e}_1 \sim \mathtt{e}_2$ to denote that terms $\mathtt{e}_1$ and $\mathtt{e}_2$ are equal up to equivalence of types.*

$$\mathtt{x} \sim \mathtt{x}$$

$$\frac{\tau \equiv_\beta \tau' \qquad \mathtt{e} \sim \mathtt{e}'}{(\lambda\mathtt{x}{:}\tau.\mathtt{e}) \sim (\lambda\mathtt{x}{:}\tau'.\mathtt{e}')} \qquad \frac{\mathtt{e}_1 \sim \mathtt{e}_1' \qquad \mathtt{e}_2 \sim \mathtt{e}_2'}{(\mathtt{e}_1\ \mathtt{e}_2) \sim (\mathtt{e}_2'\ \mathtt{e}_2')}$$

$$\frac{\mathtt{e} \sim \mathtt{e}'}{(\Lambda\alpha{:}\kappa.\mathtt{e}) \sim (\Lambda\alpha{:}\kappa.\mathtt{e}')} \qquad \frac{\mathtt{e} \sim \mathtt{e}' \qquad \tau \equiv_\beta \tau'}{(\mathtt{e}\ \tau) \sim (\mathtt{e}'\ \tau')}$$

Now we can formally state that our quoter is injective up to equivalence of types.

Theorem 6.3. *If $\langle\rangle \vdash \mathtt{e}_1 : \tau$, and $\langle\rangle \vdash \mathtt{e}_2 : \tau$, then $\mathtt{e}_1 \sim \mathtt{e}_2$ if and only if $\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}$.*

7. Operations

Our suite of operations is given in Figure 4. It consists of a self-interpreter `unquote`, a simple intensional predicate `isAbs`, a size measure `size`, a normal-form checker `nf`, and a continuation-passing-style transformation `cps`. Our suite extends those of each previous work on typed self-representation[8, 29]. Rendel et al. define a self-interpreter and a size measure, while in previous work we defined a self-interpreter, the intensional predicate `isAbs`, and a CPS transformation. Our normal-form checker is the first for a typed self-representation.

Each operation is defined using a function `foldExp` for programming folds. We also define encodings of booleans, pairs of booleans, and natural numbers that we use in our operations. We use a declaration syntax for types and terms. For example, the term declaration $\mathtt{x} : \tau = \mathtt{e}$ asserts that `e` has the type τ (i.e. $\langle\rangle \vdash \mathtt{e} : \tau$ is derivable), and substitutes `e` for `x` (essentially inlining `x`) in the subsequent declarations. We have machine checked the type of each declaration.

We give formal semantic correctness proofs for four of our operations: `unquote`, `isAbs`, `size`, and `nf`. The proofs demonstrate qualitatively that our representation is not only expressive but also easy to reason with. In the remainder of this section we briefly discuss the correctness theorems.

Each operation has a type of the form $\forall\alpha{:}\mathtt{U}.\ \mathtt{Exp}\ \alpha \to \mathtt{Op}\ \mathtt{R}\ \alpha$ for some type function `R`. When α is instantiated with a type representation $\overline{\tau}$, the result type $\mathtt{Op}\ \mathtt{R}\ \overline{\tau}$ is an interpretation under `R`:

Theorem 7.1. $\mathtt{Op}\ \mathtt{R}\ \overline{\tau} \equiv \mathtt{R}\ ([\![\tau]\!][\mathtt{F} := \mathtt{R}])$.

Each operation is defined using the function `foldExp` that constructs a fold over term representations. An interpretation of a term is obtained by substituting the designated variables `F`, `abs`, `app`, `tabs`, and `tapp` with the case functions that define an operation. The following theorem states that a fold constructed by `foldExp` maps representations to interpretations:

Theorem 7.2. *If $\mathtt{f} = \mathtt{foldExp}\ \mathtt{R}\ \mathtt{abs}'\ \mathtt{app}'\ \mathtt{tabs}'\ \mathtt{tapp}'$, and $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, then $\mathtt{f}\ \overline{\tau}\ \overline{\mathtt{e}}. \longrightarrow^* (\mathtt{q}[\mathtt{F}{:=}\mathtt{R},\ \mathtt{abs}{:=}\mathtt{abs}',\ \mathtt{app}{:=}\mathtt{app}',\ \mathtt{tabs}{:=}\mathtt{tabs}',\ \mathtt{tapp}{:=}\mathtt{tapp}'])$.*

unquote. Our first operation on term representations is our self-interpreter `unquote`, which recovers a term from its representation. Its results have types of the form `Op Id` $\overline{\tau}$. The type function `Id` is the identity function, and the operation `Op Id` recovers a type from its representation.

Theorem 7.3. *If $\Gamma \vdash \tau : *$, then $\mathtt{Op}\ \mathtt{Id}\ \overline{\tau} \equiv \tau$.*

Notice that `unquote` has the polymorphic type $(\forall\alpha{:}\mathtt{U}.\ \mathtt{Exp}\ \alpha \to \mathtt{Op}\ \mathtt{Id}\ \alpha)$. The type variable α ranges over representations of types, and the result type $\mathtt{Op}\ \mathtt{Id}\ \alpha$ recovers the type α represents. Thus, when α is instantiated with a concrete type representation $\overline{\tau}$, we get the type $\mathtt{Exp}\ \overline{\tau} \to \tau$.

Theorem 7.4. *If $\langle\rangle \vdash \mathtt{e} : \tau$, then $\mathtt{unquote}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{e}$.*

isAbs. Our second operation `isAbs` is a simple intensional predicate that checks whether its input represents an abstraction or an application. It returns a boolean on all inputs. Its result types are interpretations under `KBool`, the constant `Bool` function. The interpretation of any type under `KBool` is equivalent to `Bool`:

Theorem 7.5. *If $\Gamma \vdash \tau : *$, then $\mathtt{Op}\ \mathtt{KBool}\ \overline{\tau} \equiv \mathtt{Bool}$.*

Theorem 7.6. *Suppose $\langle\rangle \vdash \mathtt{e} : \tau$. If `e` is an abstraction then $\mathtt{isAbs}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{true}$. Otherwise `e` is an application and $\mathtt{isAbs}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{false}$.*

size. Our third operation `size` measures the size of its input representation. Its result types are interpretations under `KNat`, the constant `Nat` function. The interpretation of any type under `KNat` is equivalent to `Nat`:

Theorem 7.7. *If $\Gamma \vdash \tau : *$, then $\mathtt{Op}\ \mathtt{KNat}\ \overline{\tau} \equiv \mathtt{Nat}$.*

The size of a term excludes the types. We formally define the size of a term in order to state the correctness of `size`.

Definition 7.1. *The size of a term `e`, denoted `|e|`, is defined as:*

$$\begin{aligned}
|\mathtt{x}| &= 1\\
|\lambda\mathtt{x}{:}\tau.\mathtt{e}| &= 1 + |\mathtt{e}|\\
|\mathtt{e}_1\ \mathtt{e}_2| &= 1 + |\mathtt{e}_1| + |\mathtt{e}_2|\\
|\Lambda\alpha{:}\kappa.\mathtt{e}| &= 1 + |\mathtt{e}|\\
|\mathtt{e}\ \tau| &= 1 + |\mathtt{e}|
\end{aligned}$$

The results of `size` are Church encodings of natural numbers. We define a type `Nat` and a `zero` element and a successor function `succ`. We use the notation $\mathtt{church}_n$ to denote the Church-encoding of the natural number n. For example, $\mathtt{church}_0$ = `zero`, $\mathtt{church}_1$ = `succ zero`, $\mathtt{church}_2$ = `succ (succ zero)`, and so on.

Theorem 7.8.
If $\langle\rangle \vdash \mathtt{e} : \tau$ and $|\mathtt{e}|{=}n$, then $\mathtt{size}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{church}_n$

nf. Our fourth operation `nf` checks whether its input term is in β-normal form. Its results have types that are interpretations under `KBools`, the constant `Bools` function, where `Bools` is the type of pairs of boolean values.

Theorem 7.9. *If $\Gamma \vdash \tau : *$, then $\mathtt{Op}\ \mathtt{KBools}\ \overline{\tau} \equiv \mathtt{Bools}$.*

We program `nf` in two steps: first, we compute a pair of booleans by folding over the input term. Then we return the first component of the pair. The first boolean encodes whether a term is β-normal. The second encodes whether a term is normal and *neutral*. Intuitively, a neutral term is one that can be used in function position of an application without introducing a redex. We provide a formal definition of normal and neutral in the Appendix.

```
Bool  : *    = ∀α:*. α → α → α
true  : Bool = Λα:*. λt:α. λf:α. t
false : Bool = Λα:*. λt:α. λf:α. f
and   : Bool → Bool → Bool =
  λb1:Bool. λb2:Bool. Λα:*. λt:α. λf:α.
  b1 α (b2 α t f) f

Bools : * = ∀α:*. (Bool → Bool → α) → α
bools : Bool → Bool → Bools =
  λb1:Bool. λb2:Bool.
  Λα:*. λf:Bool → Bool → α. f b1 b2
fst : Bools → Bool =
  λbs:Bools. bs Bool (λb1:Bool. λb2:Bool. b1)
snd : Bools → Bool =
  λbs:Bools. bs Bool (λb1:Bool. λb2:Bool. b2)

Nat  : *   = ∀α:*. α → (α → α) → α
zero : Nat = Λα:*. λz:α. λs:α → α. z
succ : Nat → Nat =
  λn:Nat. Λα:*. λz:α. λs:α → α. s (n α z s)
plus : Nat → Nat → Nat =
  λm:Nat. λn:Nat. m Nat n succ
```

Definitions and operations of `Bool`, `Bools`, and `Nat`.

```
foldExp : (∀F:∗ → ∗.
           Abs F → App F → TAbs F → TApp F →
           ∀α:U. Exp α → Op F α) =
  ΛF:∗ → ∗.
  λabs  : Abs F.  λapp  : App F.
  λtabs : TAbs F. λtapp : TApp F.
  Λα:U. λe:Exp α. e F abs app tabs tapp
```

Implementation of `foldExp`

```
Id  : ∗ → ∗ = λα:∗.α

unAbs : Abs Id = Λα:∗.Λβ:∗.λf:α→β.f
unApp : App Id = Λα:∗.Λβ:∗.λf:α→β.λx:α.f x
unTAbs : TAbs Id = Λα:∗.λs:Strip Id α.λf:α.f
unTApp : TApp Id = Λα:∗.λf:α.Λβ:∗.λg:α→β.g f

unquote : (∀α:U. Exp α → Op Id α) =
  foldExp Id unAbs unApp unTAbs unTApp
```

Implementation of `unquote`

```
KBool : ∗ → ∗ = λα:∗. Bool

isAbsAbs : Abs KBool =
  Λα:∗. Λβ:∗. λf:Bool → Bool. true
isAbsApp : App KBool =
  Λα:∗. Λβ:∗. λf:Bool. λx:Bool. false
isAbsTAbs : TAbs KBool =
  Λα:∗. λstrip:Strip KBool α. λf:α. true
isAbsTApp : TApp KBool =
  Λα:∗. λf:Bool. Λβ:∗. λinst:α → Bool. false

isAbs : (∀α:U. Exp α → Bool) =
  foldExp KBool isAbsAbs isAbsApp
                isAbsTAbs isAbsTApp
```

Implementation of `isAbs`.

```
KNat : ∗ → ∗ = λα:∗. Nat

sizeAbs : Abs KNat =
  Λα:∗. Λβ:∗. λf:Nat → Nat. succ (f (succ zero))
sizeApp : App KNat =
  Λα:∗. Λβ:∗. λf:Nat. λx:Nat. succ (plus f x)
sizeTAbs : TAbs KNat =
  Λα:∗. λstrip:Strip KNat α. λf:α.
  succ (strip Nat (Λα:∗. λx:Nat. x) f)
sizeTApp : TApp KNat =
  Λα:∗. λf : Nat. Λβ:∗. λinst:α → Nat. succ f

size : (∀α:U. Exp α → Nat) =
  foldExp KNat sizeAbs sizeApp sizeTAbs sizeTApp
```

Implementation of `size`.

```
KBools : ∗ → ∗ = λα:∗. Bools

nfAbs : Abs KBools =
  Λα:∗. Λβ:∗. λf:Bools → Bools.
  bools (fst (f (bools true true))) false
nfApp : App KBools =
  Λα:∗. Λβ:∗. λf:Bools. λx:Bools.
  bools (and (snd f) (fst x)) (and (snd f) (fst x))
nfTAbs : TAbs KBools =
  Λα:∗. λstrip:Strip KBools α. λf:α.
  bools (fst (strip Bools (Λα:∗.λx:Bools.x) f))
        false
nfTApp : TApp KBools =
  Λα:∗. λf:Bools. Λβ:∗. λinst:(α → Bools).
  bools (snd f) (snd f)

nf : (∀α:U. Exp α → Bool) =
  Λα:U. λe:Exp α.
  fst (foldExp KBools nfAbs nfApp nfTAbs nfTApp e)
```

Implementation of `nf`.

```
Ct  : ∗ → ∗ = λα:∗. ∀β:∗. (α → β) → β
CPS : U → ∗ = Op Ct

cpsAbs : Abs Ct =
  Λα:∗. Λβ:∗. λf:(Ct α → Ct β).
  ΛV:∗. λk : (Ct α → Ct β) → V.
  k f
cpsApp : App Ct =
  Λα:∗. Λβ:∗. λf:Ct (Ct α → Ct β). λx:Ct α.
  ΛV:∗. λk:β → V.
  f V (λg:Ct α → Ct β. g x V k)
cpsTAbs : TAbs Ct =
  Λα:∗. λstrip:Strip Ct α. λf: α.
  ΛV:∗. λk:α → V.
  k f
cpsTApp : TApp Ct =
  Λα:∗. λf: Ct α.
  Λβ:∗. λinst:α → Ct β.
  ΛV:∗. λk:β → V.
  f V (λe:α. inst e V k)

cps : (∀α:U. Exp α → CPS α) =
  foldExp Ct cpsAbs cpsApp cpsTAbs cpsTApp
```

Implementation of `cps`.

Figure 4: Five operations on representations of F_ω terms.

Theorem 7.10. *Suppose* $\langle\rangle \vdash \mathtt{e} : \tau$.

1. *If* $\mathtt{e}$ *is* β*-normal, then* $\mathtt{nf}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^{*} \mathtt{true}$.
2. *If* $\mathtt{e}$ *is not* β*-normal, then* $\mathtt{nf}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^{*} \mathtt{false}$.

cps. Our fifth operation `cps` is a call-by-name continuation-passing-style transformation. Its result types are interpretations under `Ct`. We have also implemented a call-by-value CPS transformation, though we omit the details because it is rather similar to our call-by-name CPS. We do not formally prove the correctness of our CPS transformation. However, being defined in F_ω it is guaranteed to terminate for all inputs, and the types of the case functions provide some confidence in its correctness. Below, we show the result of applying `cps` to each of the example representations from Section 6. To aid readability, we use $[\![\tau]\!]_{\mathtt{Ct}}$ to denote $[\![\tau]\!]\mathtt{[F := Ct]}$.

The first example is the polymorphic identity function $\Lambda\alpha{:}{*}.\lambda \mathtt{x}{:}\alpha.\mathtt{x}$:

$$
\begin{array}{l}
\quad\mathtt{cps}\ \overline{\forall\alpha{:}{*}.\ \alpha \to \alpha}\ \overline{\Lambda\alpha{:}{*}.\lambda\mathtt{x}{:}\alpha.\mathtt{x}} \\
\equiv_\beta \mathtt{cpsTAbs}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ \mathtt{strip}_{\mathtt{Ct},[\![\forall\alpha:*.\alpha\to\alpha]\!]} \\
\qquad (\Lambda\alpha{:}{*}.\ \mathtt{cpsAbs}\ \alpha\ \alpha\ (\lambda\mathtt{x}{:}\mathtt{Ct}\ \alpha.\ \mathtt{x})) \\
\equiv_\beta \Lambda\beta_1 : {*}. \\
\quad \lambda\mathtt{k}_1 : \mathtt{Ct}\ (\forall\alpha{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \alpha \to \mathtt{Ct}\ \alpha)) \to \beta_1. \\
\quad \mathtt{k}_1\ (\Lambda\alpha{:}{*}.\ \Lambda\beta_2{:}{*}. \\
\qquad\quad \lambda\mathtt{k}_2 : (\mathtt{Ct}\ \alpha \to \mathtt{Ct}\ \alpha) \to \beta_2. \\
\qquad\quad \mathtt{k}_2\ (\lambda\mathtt{x} : \mathtt{Ct}\ \alpha.\ \mathtt{x}))
\end{array}
$$

The second example applies a variable of type $\forall\alpha{:}{*}.\alpha \to \alpha$ to itself:

$$
\begin{array}{l}
\quad\mathtt{cps}\ \overline{(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)} \\
\qquad\quad \overline{\lambda\mathtt{x}{:}(\forall\alpha{:}{*}.\ \alpha \to \alpha).\mathtt{x}\ (\forall\alpha{:}{*}.\ \alpha \to \alpha)\ \mathtt{x}} \\
\equiv_\beta \mathtt{cpsAbs}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}} \\
\qquad (\lambda\mathtt{x}{:}\ \mathtt{Ct}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}. \\
\qquad\ \ \mathtt{cpsApp}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}} \\
\qquad\qquad (\mathtt{cpsTApp}\ [\![\forall\alpha{:}{*}.\ \alpha \to \alpha]\!]_{\mathtt{Ct}}\ \mathtt{x} \\
\qquad\qquad\qquad [\![(\forall\alpha{:}{*}.\ \alpha \to \alpha) \to (\forall\alpha{:}{*}.\ \alpha \to \alpha)]\!]_{\mathtt{Ct}} \\
\qquad\qquad\qquad \mathtt{inst}_{[\![\forall\alpha:*.\alpha\to\alpha]\!],[\![\forall\alpha:*.\alpha\to\alpha]\!]}) \\
\qquad\qquad \mathtt{x}) \\
\equiv_\beta \Lambda\beta_1 : {*}. \\
\quad \lambda\mathtt{k}_1 : (\mathtt{Ct}\ (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a})) \to \\
\qquad\qquad \mathtt{Ct}\ (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a}))) \to \beta_1. \\
\quad \mathtt{k}_1\ (\\
\quad\ \ \lambda\mathtt{x} : \mathtt{Ct}\ (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a})). \\
\quad\ \ \Lambda\beta_2 : {*}. \\
\quad\ \ \lambda\mathtt{k}_2 : (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a})) \to \beta_2. \\
\quad\ \ (\mathtt{x}\ \beta_2\ (\lambda\mathtt{e} : \forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a}). \\
\qquad\qquad \mathtt{e}\ (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a}))\ \beta_2 \\
\qquad\qquad\ \ (\lambda\mathtt{g} : \mathtt{Ct}\ (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a})) \to \\
\qquad\qquad\qquad\quad \mathtt{Ct}\ (\forall\mathtt{a}{:}{*}.\ \mathtt{Ct}\ (\mathtt{Ct}\ \mathtt{a} \to \mathtt{Ct}\ \mathtt{a})). \\
\qquad\qquad\quad\ \mathtt{g}\ \mathtt{x}\ \beta_2\ \mathtt{k}_2))))
\end{array}
$$

8. Experiments

We have validated our techniques using an implementation of F_ω in Haskell, consisting of a parser, type checker, evaluator, β-equivalence checker, and our quoter. Each operation has been programmed, type checked, and tested. We have also confirmed that the representation of each operation type checks with the expected type.

Each of our operations are self-applicable, meaning it can be applied to a representation of itself. We have checked that the self-application of each operation type checks with the expected type. Further, we have checked that the self-application of `unquote` is β-equivalent to itself:

$$
\begin{array}{l}
\quad \mathtt{unquote}\ \overline{(\forall\alpha{:}\mathtt{U}.\ \mathtt{Exp}\ \alpha \to \mathtt{Op}\ \mathtt{Id}\ \alpha)}\ \overline{\mathtt{unquote}} \\
\equiv_\beta \mathtt{unquote}
\end{array}
$$

Our implementation is available from the web site accompanying the paper[1].

9. Discussion

Terminology. The question of whether any language (strongly normalizing or not) supports self-interpretation depends fundamentally on how one defines "representation function" and "interpreter". There are two commonly used definitions of "interpreter". The first is a function that maps the representation of a term to the term's value (i.e. the left-inverse of a particular representation function), like `eval` in JavaScript and Lisp [2, 4, 7, 9, 13, 21, 23, 26, 29]. For clarity, we will sometimes refer to this as an unquoter, since the representation function is often called a quoter. Our self-interpreter in particular is an unquoter. The other possible definition is a function that maps a representation of a term to the representation of its value. This is sometimes called an interpreter [25, 30] but also sometimes differentiated from a self-interpreter: Mogensen calls this a self-reducer [23], Berarducci and Böhm call it a reductor [7], Jay and Palsberg call it a self-enactor [18].

Some qualitative distinctions between interpreters can also be made, which might also affect the possibility of self-interpretation. A notable example is whether an interpreter is meta-circular. Unfortunately, the term "meta-circular" also has multiple meanings. Reynolds defines a meta-circular interpreter to be one which "defines each feature of the defined language by using the corresponding feature of the defining language" [30]. Abelson and Sussman state "an evaluator that is written in the same language that it evaluates is said to be metacircular" [2]. Here "evaluator" is used to mean an unquoter. Reynolds' definition allows for meta-circular interpreters that are not self-interpreters, and self-interpreters that are not meta-circular. According to Abelson and Sussman, all self-interpreters are meta-circular and vice versa. Our self-interpreter is meta-circular according to both definitions.

Many different representation schemes have been used to define interpreters. Terms can be represented as numbers, S-expressions, Church-encodings, or some user-defined data type. With respect to the treatment of variables, there is first-order, higher-order, and parametric higher-order abstract syntax. For representations of statically typed languages, use of typed representation ensures that only well-typed terms can be represented. Typed representation uses a family of types for representations, indexed by the type of the represented term, while untyped representation uses a single type for all representations. We use a typed parametric higher-order abstract syntax representation based on Church encoding. As far as we know, all unquoters defined in statically typed meta-languages are based on typed representation. Indeed, typed representation seems to be required to define an unquoter in a statically typed meta-language.

There are some properties common to all of these representation schemes: the representation function must be total (all legal terms can be represented) and injective (two terms are identical if and only if their representations are), and must produce data (e.g. normal forms). These are the requirements we have used in this paper. It is possible to strengthen the definition further. For example, we might want to require that a representation be deep. In such a case, only our deep representation would qualify as a representation.

Deep and Shallow Representation. We use the terms deep and shallow to differentiate representations supporting multiple operations (interpretations) from those supporting only one. This is analogous to deep versus shallow *embeddings* [16], but we emphasize a key difference between representation and embedding: a representation is required to be a normal form. This is generally not required of embeddings; indeed, shallow embeddings typically do not produce normal forms. A shallow embedding translates a term in one language into its interpretation defined in another language. In summary, a shallow representation *supports* one interpretation, while a shallow embedding *is* one interpretation.

Every language trivially supports shallow self-embedding, but the same is not true for shallow self-representation. For example, a shallow self-embedding for the simply-typed lambda calculus can simply map each term to itself. This is not a self-representation because the result may not be a normal form. The shallow self-representation in Section 3.3 relies on polymorphism, so it would not would work for simply typed lambda calculus.

Limitations. There are some limits to the operations we can define on our representation. For example, we cannot define our representation functions (Figure 3) in F_ω itself. This is necessarily so because the representation functions are intensional and F_ω is extensional. Stump [34] showed that it is possible to define a self-representation function within a λ-calculus extended with some intensional operations. There is a trade-off between extensional and intensional calculi: intensionality can support more expressive meta-programming, but extensionality is important for semantic properties like parametricity.

Another limitation is that representations need to be statically type checked, which limits dynamic generation of representations. For example, it is unlikely we could implement a "dynamic type checker" that maps an untyped representation to a typed representation. Something similar may be possible using stages interleaved with type checking, where an untyped representation in one stage becomes a typed representation in the next. Kiselyov calls this "Metatypechecking" [19].

Universe Hierarchies. Our techniques can be used for self-representation of other strongly-normalizing calculi more powerful than F_ω. For example, we conjecture that using kind-instantiation functions could enable a deep self-representation of F_ω^+. We would only need to use the extensional approach for representing kind polymorphism, since the kind polymorphism of F_ω^+ would enable the intensional approach to representing type polymorphism. More generally, our techniques could be used to represent a language with a hierarchy of n universes of types, with lowest universe being impredicative and the others predicative. We could use the intensional approach for the lowest $n-1$ universes, and tie the knot of self-representation by using the extensional approach for the n^{th} universe.

Type Equivalence. Our formalization of F_ω supports type conversion based on β-equivalence. In other words, two types are considered equivalent if they are beta-equivalent. This is standard – Barendregt[5] and Pierce[28] each use β-equivalence as well. Our representation would also work with a type conversion rule based on β, η-equivalence. Our key theorems 6.1 and 6.2 would be unaffected, and Theorem 6.3 would also hold assuming we update Definition 6.1 (equality of terms up to equivalence of types) accordingly.

Injective Representation Function. Intuitively, our result shows that our quoter is injective because it has a left inverse unquote. In practice, we proved injectivity in Theorem 6.3 before we went on to define unquote in Section 7. The reason is that we used injectivity to prove the correctness of unquote. We leave to future work to first define and prove correctness of unquote and then use that to prove that our quoter is injective.

10. Related Work

Typed Self-Interpretation. Pfenning and Lee [26] studied self-interpretation of Systems F and F_ω. They concluded that it seemed to be impossible for each language, and defined representations and unquoters of System F in F_ω and of F_ω in F_ω^+. They used the intensional approach to representing polymorphism that discussed in Section 4.

Rendel, et al. [29] presented the first typed self-representation and self-interpreter. Their language System F_ω^* extends F_ω with a `Type:Type` rule that unifies the levels of types and kinds. As a result, F_ω^* is not strongly-normalizing, and type checking is undecidable. Their representation also used the intensional approach to representing polymorphism. They presented two operations, an unquoter and a size measure. Their implementation of `size` relied on a special $\perp$ type to strip redundant quantifiers. The type $\perp$ inhabits every kind, but is not used to type check terms. We strip redundant quantifiers using special instantiation functions that are generated by the quoter.

Jay and Palsberg [18] presented a typed self-representation and self-interpreter for a combinator calculus, with a λ-calculus surface syntax. Their calculus had undecidable type checking and was not strongly normalizing.

In previous work [8] we presented a typed self-representation for System U, which is not strongly normalizing but does have decidable type checking. This was the first self-representation for a language with decidable type checking. The representation was similar to those of Pfenning and Lee [26] and Rendel, et al. [29] and also used the intensional approach to representing polymorphism. We presented three operations on the representation of System U terms – `unquote`, `isAbs`, and `cps`. Not all System U kinds are inhabited, so redundant quantifiers couldn't be stripped. This prevented operations like `size` or `nf`. We also represented System U types of kind $*$, but did not have a substitution theorem like Theorem 5.2. As a result, the representation of a type application could have the wrong type, which we corrected using a kind of coercion. Our representation of F_ω types is designed to avoid the need for such coercions, which simplifies our representation and the proofs of our theorems.

Representation Technique. We mix standard representation techniques with a minimal amount of novelty needed to tie the knot of self-representation. At the core is a typed Higher-order Abstract Syntax (HOAS) based on Church encoding. Similar representations were used in previous work on typed representation [8, 9, 26, 29].

Our previous work [8] showed self-representation is possible using only the intensional approach to representing polymorphism requires and two impredicative universes (the types and kinds of System U). Our extensional approach presented here allows us to use only a single impredicative universe (the types of F_ω).

The shallow representation of System F also requires impredicativity to block type applications. We leave the question whether self-representation is possible without any impredicativity for future work.

Typed Meta-Programming. Typed self-interpretation is a particular instance of typed meta-programming, which involves a typed representation of one language in a possibly different language, and operations on that representation. Typed meta-programming has been studied extensively, and continues to be an active research area. Chen and Xi [11, 12] demonstrated that types can make meta-programming less error-prone.

Carette et al. [9] introduced tagless representations, which are more efficient than other techniques and use simpler types. Our representation is also tagless, though we use ordinary λ-abstractions to abstract over the case functions of an operation, while they use Haskell type classes or OCaml modules. The object languages they represented did not include polymorphism. Our extensional technique could be used to program tagless representations of polymorphic languages in Haskell or OCaml.

MetaML [35] supports *generative* typed meta-programming for multi-stage programming. It includes a built-in unquoter, while we program `unquote` as a typed F_ω term.

Trifonov et al. [36] define a language with fully reflexive intensional type analysis, which supports type-safe runtime type introspection. Instead of building representations of types, their language includes special operators to support iterating over types. They programmed generic programs like marshalling values for transmission over a network. Generic programming and meta-programming are different techniques: generic programs operate on programs or program values, and meta-programs operate on representations of programs. These differences mean that each technique is suited to some problems better than the other.

Dependently-Typed Representation. Some typed representations use dependent types to ensure that only well-typed terms can be represented. For example, Harper and Licata [17] represented simply-typed λ-calculus in LF, and Schürmann et al. [31] represented F_ω in LF. Chapman [10] presented a meta-circular representation of a dependent type theory in Agda. These representations are quite useful for *mechanized meta-theory* – machine-checked proofs of the meta-theorems for the represented language. The demands of mechanized metatheory appear to be rather different from those of self-interpretation. It is an open question whether a dependently-typed self-representation can support a self-interpreter.

Dependent Types. Dependent type theory is of particular interest among strongly-normalizing languages, as it forms the basis of proof assistants like Coq and Agda. While dependent type theory generally includes dependent sum and product types, modern variants also support inductive definitions, an infinite hierarchy of universes, and universe polymorphism. A self-representation of such a language would need to represent all of these features, each of which comes with its own set of challenges.

Altenkirch and Kaposi [3] formalize a simple dependent type theory in another type theory (Agda extended with some postulates). They focus on the key problem of defining a typed representation of dependent type theory: that the types, terms, type contexts, and type equality are all mutually-dependent. Their solution relies on Quotient-Inductive Types (QITs), a special case of Higher-Inductive Types from Homotopy Type Theory. Their work is an important step towards a self-representation of dependently type theory. To achieve full self-representation, one would need to represent QITs themselves, which the authors cite as an open challenge.

Untyped Representation. The literature contains many examples of untyped representations for typed languages, including for Coq [6] and Haskell [25]. Untyped representations generally use a single type like `Exp` to type check all representations, and permit ill-typed terms to be represented. Template Haskell [32] uses an untyped representation and supports user-defined operations on representations. Since representations are not guaranteed to be well-typed by construction, generated code needs to be type checked.

Coercions. Our instantiation functions are similar to coercions or *retyping functions*: they change the type of a term without affecting its behavior. Cretin and Rémy [15] studied erasable coercions for System F_η [22], including coercions that perform instantiations. We conjecture that our self-representation technique would work for an extension of F_ω with erasable instantiation coercions, and that these coercions could replace instantiation functions in our extensional approach to representing polymorphism. This could provide some advantages over the instantiation functions used in this paper. In particular, a weakness of instantiation functions is that their types overlap with those of other terms. Therefore, it is possible to use something other than instantiation function (e.g. a constant function) where one is expected. As a result, we can write a closed term of type `Exp` τ (for some τ) that is not the representation of any term. The types of Cretin and Rémy's coercions do not overlap with the types of terms, so replacing instantiation functions with instantiation coercions could eliminate this problem.

11. Conclusion

We have solved two open problems posed by Pfenning and Lee. First, we have defined a shallow self-representation technique that supports self-interpretation for each of System F and System F_ω. Second, we have defined a deep self-representation for System F_ω that supports a variety of operations including a self-interpreter.

Our result is consistent with the classical theorem that the universal function for the total computable functions cannot be total. The reason is that the theorem assumes that terms are represented as numbers using Gödel numbering. We show that a typed representation can ensure that the diagonalization gadget central to the proof fails to type check.

Our result opens the door to self-representations and self-interpreters for other strongly normalizing languages. Our techniques create new opportunities for type-checking self-applicable meta-programs, with potential applications in typed macro systems, partial evaluators, compilers, and theorem provers.

Some open questions include:

- Is a self-reducer possible in a strongly normalizing language?
- Is it possible to define a self-interpreter or self-reducer using a first-order representation (for example, based on SK combinators) in a strongly normalizing language?

Acknowledgments. We thank John Bender, Neal Glew, Bob Harper, Todd Millstein, and the POPL reviewers for helpful comments, discussions, and suggestions. This material is based upon work supported by the National Science Foundation under Grant Number 1219240.

References

[1] The webpage accompanying this paper is available at http://compilers.cs.ucla.edu/popl16/. The full paper with the appendix is available there, as is the source code for our implementation of System F_ω and our operations.

[2] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT electrical engineering and computer science series. MIT Press, 1987.

[3] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[4] Henk Barendregt. Self-interpretations in lambda calculus. *J. Funct. Program*, 1(2):229–233, 1991.

[5] HP Barendregt. *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures: Abramski, S. (ed)*, chapter Lambda Calculi with Types. Oxford University Press, Inc., New York, NY, 1993.

[6] Bruno Barras and Benjamin Werner. Coq in coq. Technical report, 1997.

[7] Alessandro Berarducci and Corrado Böhm. A self-interpreter of lambda calculus having a normal form. In *CSL*, pages 85–99, 1992.

[8] Matt Brown and Jens Palsberg. Self-Representation in Girard's System U. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 471–484, New York, NY, USA, 2015. ACM.

[9] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[10] James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.

[11] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 275–286, Uppsala, Sweden, August 2003.

[12] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. *Journal of Functional Programming*, 15(6):797–835, 2005.

[13] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[14] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[15] Julien Cretin and Didier Rémy. On the power of coercion abstraction. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 361–372, New York, NY, USA, 2012. ACM.

[16] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM.

[17] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, July 2007.

[18] Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of ICFP'11, ACM SIGPLAN International Conference on Functional Programming*, pages 247–258, Tokyo, September 2011.

[19] Oleg Kiselyov. Metatypechecking: Staged typed compilation into gadt using typeclasses. http://okmij.org/ftp/tagless-final/tagless-typed.html#tc-GADT-tc.

[20] Stephen C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, pages 340–353, 1936.

[21] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[22] John C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, February 1988.

[23] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992. See also DIKU Report D–128, Sep 2, 1994.

[24] Greg Morrisett. F-omega – the workhorse of modern compilers. http://www.eecs.harvard.edu/ greg/cs256sp2005/lec16.txt, 2005.

[25] Matthew Naylor. Evaluating Haskell in Haskell. *The Monad.Reader*, 10:25–33, 2008.

[26] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.

[27] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[28] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[29] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. Typed self-representation. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, June 2009.

[30] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740. ACM Press, 1972. The paper later appeared in Higher-Order and Symbolic Computation, 11, 363–397 (1998).

[31] Carsten Schürmann, Dachuan Yu, and Zhaozhong Ni. A representation of f_ω in lf. *Electronic Notes in Theoretical Computer Science*, 58(1):79 – 96, 2001.

[32] Tim Sheard and Simon Peyton Jones. Template metaprogramming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[33] T. Stuart. *Understanding Computation: Impossible Code and the Meaning of Programs*. Understanding Computation. O'Reilly Media, Incorporated, 2013.

[34] Aaron Stump. Directly reflective meta-programming. *Higher Order Symbol. Comput.*, 22(2):115–144, June 2009.

[35] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

[36] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. *SIGPLAN Not.*, 35(9):82–93, September 2000.

[37] David Turner. Total functional programming. *Journal of Universal Computer Science*, 10, 2004.

[38] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[39] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 249–262, New York, NY, USA, 2003. ACM.

A. Proofs

A.1 Section 3 Proofs

Theorem 3.4.
If $\langle\rangle \vdash \mathtt{e} : \tau$*, then* $\langle\rangle \vdash \overline{\mathtt{e}} : (\forall\alpha{:}{*}.\ \alpha \to \alpha) \to \tau$ *and* $\overline{\mathtt{e}}$ *is in* β*-normal form.*

Proof. Straightforward. □

Theorem 3.5.
If $\langle\rangle \vdash \mathtt{e} : \tau$*, then* $\mathtt{unquote}\ \tau\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{e}$.

Proof. Straightforward. □

A.2 Section 5 Proofs

Lemma A.1. *If* $\Gamma \vdash \tau : \kappa$*, then* $\mathtt{F}{:}{*} \to {*},\Gamma \vdash [\![\tau]\!] : \kappa$.

Proof. By induction on the structure of τ. Let $\Gamma' = (\mathtt{F}{:}{*} \to {*},\Gamma)$.

Suppose τ is a type variable α. Then $[\![\tau]\!] = [\![\alpha]\!] = \alpha = \tau$, and the result is immediate.

Suppose τ is an arrow type $\tau_1 \to \tau_2$. By the inversion lemma, we have that $\kappa{=}{*}$ and $\Gamma \vdash \tau_1 : *$ and $\Gamma \vdash \tau_2 : *$. By definition, $[\![\tau]\!] = \mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$. It is sufficent to show that $\Gamma' \vdash \mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!] : *$. By induction, $\Gamma' \vdash [\![\tau_1]\!] : *$ and $\Gamma' \vdash [\![\tau_2]\!] : *$. Therefore, $\Gamma' \vdash \mathrm{F}\ [\![\tau_1]\!] \to \mathrm{F}\ [\![\tau_2]\!] : *$, as required.

Suppose τ is a quantified type $(\forall\alpha{:}\kappa.\tau_1)$. By the inversion lemma, we have that $\Gamma,\alpha{:}\kappa \vdash \tau_1 : *$. By definition, $[\![\tau]\!] = \forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]$. It is sufficient to show that $\Gamma' \vdash (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]) : *$. By induction, we have that $(\Gamma,\alpha{:}\kappa) \vdash [\![\tau_1]\!] : *$. Then $\Gamma',\alpha{:}\kappa \vdash \mathtt{F}\ [\![\tau_1]\!] : *$, so $\Gamma' \vdash (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]) : *$, as required.

Suppose that τ is a λ-abstraction $(\lambda\alpha{:}\kappa_1.\tau_1)$. By the inversion lemma, we have that $\kappa = \kappa_1 \to \kappa_2$ and $(\Gamma,\alpha{:}\kappa_1) \vdash \tau_1 : \kappa_2$. By definition, $[\![\tau]\!] = [\![\lambda\alpha{:}\kappa_1.\tau_1]\!] = \lambda\alpha{:}\kappa_1.[\![\tau_1]\!]$. It is sufficient to show that $\Gamma' \vdash (\lambda\alpha{:}\kappa_1.[\![\tau_1]\!]) : \kappa_1 \to \kappa_2$. By weakening, $(\Gamma,\alpha{:}\kappa_1) \vdash \mathtt{F} : * \to *$. Therefore, the induction hypothesis gives $(\Gamma',\alpha{:}\kappa_1) \vdash [\![\tau_1]\!] : \kappa_2$. Therefore, $\Gamma' \vdash (\lambda\alpha{:}\kappa_1.[\![\tau_1]\!]) : \kappa_1 \to \kappa_2$, as required.

Suppose τ is an application $\tau_1\ \tau_2$. By the inversion lemma, $\Gamma \vdash \tau_1 : \kappa_2 \to \kappa$ and $\Gamma \vdash \tau_2 : \kappa_2$. By definition, $[\![\tau]\!] = [\![\tau_1\ \tau_2]\!] = [\![\tau_1]\!]\ [\![\tau_2]\!]$. It is sufficient to show $\Gamma \vdash [\![\tau_1]\!]\ [\![\tau_2]\!] : \kappa$. By the induction hypothesis, $\Gamma' \vdash [\![\tau_1]\!] : \kappa_2 \to \kappa$ and $\Gamma' \vdash [\![\tau_2]\!] : \kappa_2$. Therefore, $\Gamma' \vdash [\![\tau_1]\!]\ [\![\tau_2]\!] : \kappa$ as required. □

Theorem 5.1. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \overline{\tau} : (* \to *) \to \kappa$.

Proof. Suppose $\Gamma \vdash \tau : \kappa$. By definition, we have that $\overline{\tau} = \lambda\mathtt{F}{:}{*} \to {*}.[\![\tau]\!]$. Without loss of generality via renaming, assume $\mathtt{F}$ does not occur free in τ. Then by weakening, $\Gamma,\mathtt{F}{:}{*} \to * \vdash \tau : \kappa$. By Lemma A.1, $\Gamma,\mathtt{F}{:}{*} \to * \vdash [\![\tau]\!] : \kappa$. Therefore $\Gamma \vdash \overline{\tau} : (* \to *) \to \kappa$.

□

Theorem 5.2. *For any types* τ *and* σ*, and any type variable* α*, we have* $[\![\tau]\!][\alpha := [\![\sigma]\!]] = [\![\tau[\alpha := \sigma]]\!]$.

Proof. By induction on the structure of τ.

Suppose τ is the type variable α. Then $[\![\tau]\!] = \alpha$, so $[\![\tau]\!][\alpha := [\![\sigma]\!]] = [\![\sigma]\!]$. Also $[\![\tau[\alpha := \sigma]]\!] = [\![\alpha[\alpha := \sigma]]\!] = [\![\sigma]\!]$, as required.

Suppose τ is a type variable $\beta \neq \alpha$. Then $[\![\tau]\!] = \beta$, so $[\![\tau]\!][\alpha := [\![\sigma]\!]] = \beta[\alpha := [\![\sigma]\!]] = \beta$. Also $[\![\tau[\alpha := \sigma]]\!] = [\![\beta[\alpha := \sigma]]\!] = [\![\beta]\!] = \beta$, as required.

Suppose τ is an arrow type $\tau_1 \to \tau_2$. By induction, we have that $[\![\tau_1]\!][\alpha := [\![\sigma]\!]] = [\![\tau_1[\alpha := \sigma]]\!]$ and $[\![\tau_2]\!][\alpha := [\![\sigma]\!]] = [\![\tau_2[\alpha := \sigma]]\!]$. By definition, $[\![\tau]\!] = \mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$, so $[\![\tau]\!][\alpha := [\![\sigma]\!]] = \mathtt{F}\ ([\![\tau_1]\!][\alpha := [\![\sigma]\!]]) \to \mathtt{F}\ ([\![\tau_2]\!][\alpha := [\![\sigma]\!]]) = \mathtt{F}\ [\![\tau_1[\alpha := \sigma]]\!] \to \mathtt{F}\ [\![\tau_2[\alpha := \sigma]]\!]$. Also, $[\![\tau[\alpha := \sigma]]\!] = [\![(\tau_1 \to \tau_2)[\alpha := \sigma]]\!] = [\![(\tau_1[\alpha := \sigma]) \to (\tau_2[\alpha := \sigma])]\!] = \mathtt{F}\ [\![\tau_1[\alpha := \sigma]]\!] \to \mathtt{F}\ [\![\tau_2[\alpha := \sigma]]\!]$, as required.

Suppose that τ is a quantified type $\forall\beta{:}\kappa.\tau_1$. By induction, we have that $[\![\tau_1]\!][\alpha := [\![\sigma]\!]] = [\![\tau_1[\alpha := \sigma]]\!]$. By definition, $[\![\tau]\!] = \forall\beta{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]$, so $[\![\tau]\!][\alpha := [\![\sigma]\!]] = (\forall\beta{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!])[\alpha := [\![\sigma]\!]] = \forall\beta{:}\kappa.\ \mathtt{F}\ ([\![\tau_1]\!][\alpha := [\![\sigma]\!]]) = \forall\beta{:}\kappa.\ \mathtt{F}\ [\![\tau_1[\alpha := \sigma]]\!]$. Also, $[\![\tau[\alpha := \sigma]]\!] = [\![(\forall\beta{:}\kappa.\ \tau_1)[\alpha := \sigma]]\!] = [\![\forall\beta{:}\kappa.\ (\tau_1[\alpha := \sigma])]\!] = \forall\beta{:}\kappa.\ \mathtt{F}\ [\![\tau_1[\alpha := \sigma]]\!]$, as required.

Suppose that τ is a λ-abstraction $\lambda\beta{:}\kappa.\tau_1$. By induction, we have that $[\![\tau_1]\!][\alpha := [\![\sigma]\!]] = [\![\tau_1[\alpha := \sigma]]\!]$. By definition, $[\![\tau]\!] = \lambda\beta{:}\kappa.\ [\![\tau_1]\!]$, so $[\![\tau]\!][\alpha := [\![\sigma]\!]] = (\lambda\beta{:}\kappa.\ [\![\tau_1]\!])[\alpha := [\![\sigma]\!]] = \lambda\beta{:}\kappa.\ ([\![\tau_1]\!][\alpha := [\![\sigma]\!]]) = \lambda\beta{:}\kappa.\ [\![\tau_1[\alpha := \sigma]]\!]$. Also, $[\![\tau[\alpha := \sigma]]\!] = [\![(\lambda\beta{:}\kappa.\ \tau_1)[\alpha := \sigma]]\!] = [\![\lambda\beta{:}\kappa.\ (\tau_1[\alpha := \sigma])]\!] = \lambda\beta{:}\kappa.\ [\![\tau_1[\alpha := \sigma]]\!]$, as required.

Suppose τ is an application $\tau_1\ \tau_2$. By induction, we have that $[\![\tau_1]\!][\alpha := [\![\sigma]\!]] = [\![\tau_1[\alpha := \sigma]]\!]$ and $[\![\tau_2]\!][\alpha := [\![\sigma]\!]] = [\![\tau_2[\alpha := \sigma]]\!]$. By definition $[\![\tau]\!][\alpha := [\![\sigma]\!]] = [\![\tau_1\ \tau_2]\!][\alpha := [\![\sigma]\!]] = ([\![\tau_1]\!]\ [\![\tau_2]\!])[\alpha := [\![\sigma]\!]] = ([\![\tau_1]\!][\alpha := [\![\sigma]\!]])\ ([\![\tau_2]\!][\alpha := [\![\sigma]\!]]) = [\![\tau_1[\alpha := \sigma]]\!]\ [\![\tau_2[\alpha := \sigma]]\!]$. Also $[\![\tau[\alpha := \sigma]]\!] = [\![(\tau_1\ \tau_2)[\alpha := \sigma]]\!] = [\![(\tau_1[\alpha := \sigma])\ (\tau_2[\alpha := \sigma])]\!] = [\![\tau_1[\alpha := \sigma]]\!]\ [\![\tau_2[\alpha := \sigma]]\!]$, as required. □

The following is an inductive definition of terms in normal and neutral form.

Definition A.1 (β-Normal and β-Neutral Terms)**.**

$$\frac{\mathtt{x}\ \textit{is a variable.}}{\mathtt{x}\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.}}$$

$$\frac{\mathtt{e}\ \textit{is } \beta\textit{-normal.}}{\begin{array}{c}(\lambda\mathtt{x}{:}\tau.\mathtt{e})\ \textit{is } \beta\textit{-normal.}\\ (\Lambda\alpha{:}\kappa.\mathtt{e})\ \textit{is } \beta\textit{-normal.}\end{array}}$$

$$\frac{\mathtt{e}_1\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.} \qquad \mathtt{e}_2\ \textit{is } \beta\textit{-normal.}}{\begin{array}{c}(\mathtt{e}_1\ \mathtt{e}_2)\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.}\\ (\mathtt{e}_1\ \tau)\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.}\end{array}}$$

Similarly, here is an inductive definition of β-normal and β-neutral types. Intuitively, a β-normal type contains no β-redexes.

Definition A.2 (β-Normal and β-Neutral Types)**.**

$$\frac{\alpha\ \textit{is a variable.}}{\alpha\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.}}$$

$$\frac{\tau\ \textit{is } \beta\textit{-normal.}}{(\lambda\mathtt{x}{:}\kappa.\tau)\ \textit{is } \beta\textit{-normal.}}$$

$$\frac{\tau_1\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.} \qquad \tau_2\ \textit{is } \beta\textit{-normal.}}{(\tau_1\ \tau_2)\ \textit{is } \beta\textit{-normal and } \beta\textit{-neutral.}}$$

Lemma A.2. *If* $\Gamma \vdash \tau : \kappa$*, then there exists a* β*-normal* τ' *such that* $\Gamma \vdash \tau' : \kappa$ *and* $\tau \equiv \tau'$.

Proof. Standard. □

Lemma A.3. *If* $\Gamma \vdash \tau_1 : \kappa$ *and* $\Gamma \vdash \tau_2 : \kappa$ *and* $\tau_1 \equiv \tau_2$*, then* $\tau_1 = \tau_2$.

Proof. Standard. □

Lemma A.4.

1. *If τ is normal, then $[\![\tau]\!]$ is normal.*
2. *If τ is normal and neutral, then $[\![\tau]\!]$ is normal and neutral.*

Proof. By induction on the structure of τ.

Suppose τ is a variable α. We prove cases (1) and (2) simultaneously. Since τ is variable, it is normal and neutral. We have $[\![\tau]\!] = \alpha$, which is normal and neutral.

Suppose τ is an arrow $\tau_1 \to \tau_2$. We prove cases (1) and (2) simultaneously. Suppose that τ is normal. Then $[\![\tau]\!] = \mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]$. We now show that $\mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]$ is normal and neutral. Since τ is normal, τ_1 and τ_2 are normal. By induction, $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ are normal. Since F is a variable, it is normal and neutral. Therefore $\mathsf{F}\ [\![\tau_1]\!]$ and $\mathsf{F}\ [\![\tau_2]\!]$ are normal and neutral, so $\mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]$ is normal and neutral, as required.

Suppose τ is a quantified type $\forall\alpha{:}\kappa.\tau_1$. We prove cases (1) and (2) simultaneously. Suppose that τ is normal. Then τ_1 is normal. By the induction hypothesis, $[\![\tau_1]\!]$ is normal. By definition, $[\![\tau]\!] = \forall\alpha{:}\kappa.\ \mathsf{F}\ [\![\tau_1]\!]$. Since F is a variable, it is normal and neutral, so $\forall\alpha{:}\kappa.\ \mathsf{F}\ [\![\tau_1]\!]$ is normal and neutral, as required.

Suppose τ is a λ-abstraction $\lambda\alpha{:}\kappa.\tau_1$. Then τ is not neutral, so we only consider case (1). Suppose τ is normal. Then τ_1 is normal. By the induction hypothesis, $[\![\tau_1]\!]$ is normal. By definition, $[\![\tau]\!] = \lambda\alpha{:}\kappa.[\![\tau_1]\!]$. Since $[\![\tau_1]\!]$ is normal, $[\![\tau]\!]$ is normal as required.

Suppose τ is an application $\tau_1\ \tau_2$. We consider both cases simultaneously. Suppose τ is normal. Then τ_1 is neutral and normal, and τ_2 is normal. By the induction hypothesis, $[\![\tau_1]\!]$ is neutral and normal, and $[\![\tau_2]\!]$ is normal. By definition, $[\![\tau]\!] = [\![\tau_1]\!]\ [\![\tau_2]\!]$. Since $[\![\tau_1]\!]$ is neutral and normal and $[\![\tau_2]\!]$ is normal, $[\![\tau]\!]$ is neutral and normal, as required. ☐

Definition A.3 (Normal form of a term). *Suppose* $\mathsf{e}_1 \longrightarrow^* \mathsf{e}_2$ *and* e_2 *is in normal form. Then we say that* e_2 *is the normal form of* e_1*.*

Lemma A.5. *If* $[\![\tau]\!] = [\![\sigma]\!]$*, then* $\tau = \sigma$*.*

Proof. By induction on the structure of τ.

Suppose τ is a variable x. Then $[\![\tau]\!] = \mathsf{x} = [\![\sigma]\!]$. Therefore, $\sigma = \mathsf{x}$.

Suppose τ is an arrow type $\tau_1 \to \tau_2$. Then $[\![\tau_1 \to \tau_2]\!] = \mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!] = [\![\sigma]\!]$. Therefore, $\sigma{=}(\sigma_1 \to \sigma_2)$, and $[\![\tau_1]\!] = [\![\sigma_1]\!]$, and $[\![\tau_2]\!] = [\![\sigma_2]\!]$. By induction, $\tau_1 = \sigma_1$ and $\tau_2 = \sigma_2$. Therefore, $\sigma = (\sigma_1 \to \sigma_2) = (\tau_1 \to \tau_2) = \tau$.

Suppose τ is a quantified type $\forall\alpha{:}\kappa.\tau_1$. Then $[\![\tau]\!] = [\![\forall\alpha{:}\kappa.\tau_1]\!] = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau_1]\!])$. Therefore, $[\![\sigma]\!] = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau_1]\!])$, so $\sigma = (\forall\alpha{:}\kappa.\sigma_1)$ and $[\![\tau_1]\!] = [\![\sigma_1]\!]$. By induction, $\tau_1 = \sigma_1$, so $\sigma = (\forall\alpha{:}\kappa.\sigma_1) = (\forall\alpha{:}\kappa.\tau_1) = \tau$.

Suppose τ is a λ-abstraction $(\lambda\alpha{:}\kappa.\tau_1)$. Then $[\![\tau]\!] = [\![\lambda\alpha{:}\kappa.\tau_1]\!] = (\lambda\alpha{:}\kappa.[\![\tau_1]\!])$. Therefore, $[\![\sigma]\!] = (\lambda\alpha{:}\kappa.[\![\tau_1]\!])$, so $\sigma = (\lambda\alpha{:}\kappa.\sigma_1)$, and $[\![\tau_1]\!] = [\![\sigma_1]\!]$. By induction, $\tau_1 = \sigma_1$, so $\sigma = (\lambda\alpha{:}\kappa.\sigma_1) = (\lambda\alpha{:}\kappa.\tau_1) = \tau$.

Suppose τ is a type application $(\tau_1\ \tau_2)$. Then $[\![\tau]\!] = [\![\tau_1\ \tau_2]\!] = [\![\tau_1]\!]\ [\![\tau_2]\!]$. Therefore, $[\![\sigma]\!] = [\![\tau_1]\!]\ [\![\tau_2]\!]$, so $\sigma = (\sigma_1\ \sigma_2)$ and $[\![\tau_1]\!] = [\![\sigma_1]\!]$ and $[\![\tau_2]\!] = [\![\sigma_2]\!]$. By induction, $\tau_1 = \sigma_1$ and $\tau_2 = \sigma_2$, so $\sigma = (\sigma_1\ \sigma_2) = (\tau_1\ \tau_2)$. ☐

Lemma A.6. $\tau \equiv \sigma$ *if and only if* $[\![\tau]\!] \equiv [\![\sigma]\!]$*.*

Proof. **$(\tau \equiv \sigma) \implies ([\![\tau]\!] \equiv [\![\sigma]\!])$:**

By induction on the derivation of $\tau \equiv \sigma$.

Suppose $\tau \equiv \sigma$ is by reflexivity. Then $\tau{=}\sigma$, so $[\![\tau]\!]{=}[\![\sigma]\!]$.

Suppose $\tau \equiv \sigma$ is by symmetry. Then $\sigma \equiv \tau$, and by induction $[\![\sigma]\!] \equiv [\![\tau]\!]$. Therefore, $[\![\tau]\!] \equiv [\![\sigma]\!]$ follows by symmetry.

Suppose $\tau \equiv \sigma$ is by transitivity. Then $\tau \equiv \tau'$ and $\tau' \equiv \sigma$, and by induction, $[\![\tau]\!] \equiv [\![\tau']\!]$ and $[\![\tau']\!] \equiv [\![\sigma]\!]$. Therefore, $[\![\tau]\!] \equiv [\![\sigma]\!]$ follows by transitivity.

Suppose $\tau \equiv \sigma$ is by the rule for $\to$ types. Then $\tau = \tau_1 \to \tau_2$, $\sigma = \sigma_1 \to \sigma_2$, and $\tau_1 \equiv \sigma_1$ and $\tau_2 \equiv \sigma_2$. By induction, $[\![\tau_1]\!] \equiv [\![\sigma_1]\!]$ and $[\![\tau_2]\!] \equiv [\![\sigma_2]\!]$. Therefore, $[\![\tau]\!] = [\![\tau_1 \to \tau_2]\!] = \mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]) \equiv \mathsf{F}\ [\![\sigma_1]\!] \to \mathsf{F}\ [\![\sigma_2]\!] = [\![\sigma_1 \to \sigma_2]\!] = [\![\sigma]\!]$.

Suppose $\tau \equiv \sigma$ is by the rule for $\forall$ types. Then $\tau{=}(\forall\alpha{:}\kappa.\tau_1)$ and $\sigma{=}(\forall\alpha{:}\kappa.\sigma_1)$ and $\tau_1 \equiv \sigma_1$. By induction, we have that $[\![\tau_1]\!] \equiv [\![\sigma_1]\!]$. Therefore, $[\![\tau]\!] = [\![\forall\alpha{:}\kappa.\tau_1]\!] = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau_1]\!]) = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\sigma_1]\!]) = [\![\forall\alpha{:}\kappa.\sigma_1]\!] = [\![\sigma]\!]$.

Suppose $\tau \equiv \sigma$ is by the rule for λ-abstractions. Then $\tau{=}(\lambda\alpha{:}\kappa.\tau_1)$ and $\sigma{=}(\lambda\alpha{:}\kappa.\sigma_1)$ and $\tau_1 \equiv \sigma_1$. By induction, we have that $[\![\tau_1]\!] \equiv [\![\sigma_1]\!]$. Therefore, $[\![\tau]\!] = [\![\lambda\alpha{:}\kappa.\tau_1]\!] = (\lambda\alpha{:}\kappa.[\![\tau_1]\!]) = (\lambda\alpha{:}\kappa.[\![\sigma_1]\!]) = [\![\lambda\alpha{:}\kappa.\sigma_1]\!] = [\![\sigma]\!]$.

Suppose $\tau \equiv \sigma$ is by the rule for type applications. Then $\tau{=}(\tau_1\ \tau_2)$ and $\sigma{=}(\sigma_1\ \sigma_2)$, and $\tau_1 \equiv \sigma_1$ and $\tau_2 \equiv \sigma_2$. By induction, $[\![\tau_1]\!] \equiv [\![\sigma_1]\!]$ and $[\![\tau_2]\!] \equiv [\![\sigma_2]\!]$. Therefore, $[\![\tau]\!] = [\![\tau_1\ \tau_2]\!] = [\![\tau_1]\!]\ [\![\tau_2]\!]) \equiv [\![\sigma_1]\!]\ [\![\sigma_2]\!] = [\![\sigma_1\ \sigma_2]\!] = [\![\sigma]\!]$.

Suppose $\tau \equiv \sigma$ is by the rule for α-conversion. Then $\tau{=}(\lambda\alpha{:}\kappa.\tau_1)$ and $\sigma{=}(\lambda\beta{:}\kappa.\tau_1[\alpha{:=}\beta])$. We have that $[\![\tau]\!] = [\![\lambda\alpha{:}\kappa.\tau_1]\!]{=}(\lambda\alpha{:}\kappa.[\![\tau_1]\!]) \equiv (\lambda\beta{:}\kappa.[\![\tau_1]\!][\alpha{:=}\beta]) = (\lambda\beta{:}\kappa.\ [\![\tau_1]\!][\alpha{:=}[\![\beta]\!]])$. By Theorem 5.2, $(\lambda\beta{:}\kappa.\ [\![\tau_1]\!][\alpha{:=}[\![\beta]\!]]) = (\lambda\beta{:}\kappa.[\![\tau_1[\alpha{:=}\beta]]\!]) = [\![\lambda\beta{:}\kappa.\tau_1[\alpha{:=}\beta]]\!] = [\![\sigma]\!]$.

Suppose $\tau \equiv \sigma$ is by the rule for β-reduction. Then $\tau{=}((\lambda\alpha{:}\kappa.\tau_1)\tau_2)$ and $\sigma{=}(\tau_1[\alpha{:=}\tau_2])$. We have that $[\![\tau]\!] = [\![((\lambda\alpha{:}\kappa.\tau_1)\tau_2)]\!] = ([\![(\lambda\alpha{:}\kappa.\tau_1)]\!]\ [\![\tau_2]\!]) = ((\lambda\alpha{:}\kappa.[\![\tau_1]\!])\ [\![\tau_2]\!]) \equiv ([\![\tau_1]\!][\alpha{:=}[\![\tau_2]\!]])$. By Theorem 5.2, $([\![\tau_1]\!][\alpha{:=}[\![\tau_2]\!]]) = [\![\tau_1[\alpha{:=}\tau_2]]\!] = [\![\sigma]\!]$.

$([\![\tau]\!] \equiv [\![\sigma]\!]) \implies (\tau \equiv \sigma)$:

Suppose $([\![\tau]\!] \equiv [\![\sigma]\!])$. By Lemma A.2, there exist β-normal types τ' and σ' such that $\tau \equiv_\beta \tau'$ and $\sigma \equiv_\beta \sigma'$. By (1) $(\tau \equiv \sigma) \implies ([\![\tau]\!] \equiv [\![\sigma]\!])$, we have that $[\![\tau]\!] \equiv [\![\tau']\!]$ and $[\![\sigma]\!] \equiv [\![\sigma']\!]$. By transitivity, we have that $[\![\tau']\!] \equiv [\![\sigma']\!]$. Therefore, it is sufficient to show that $\tau' \equiv \sigma'$. By Lemma A.4, $[\![\tau']\!]$ and $[\![\sigma']\!]$ are normal. Therefore, Lemma states that $[\![\tau']\!] = [\![\sigma']\!]$. By Lemma A.5, $\tau' = \sigma'$. Therefore, $\tau \equiv \sigma$. ☐

Theorem 5.3. $\tau \equiv \sigma$ *if and only if* $\overline{\tau} \equiv \overline{\sigma}$*.*

Proof. Follows from Lemma A.6. ☐

Definition A.4 (Instances of a quantified type). *We say that a type* σ *is an instance of a quantified type* τ *if and only if* τ = $(\forall\alpha{:}\kappa.\tau_1)$*, and* σ = $(\tau_1[\alpha{:=}\tau_2])$*, and there exists an environment* Γ *such that* $\Gamma \vdash (\forall\alpha{:}\kappa.\tau_1) : *$ *and* $\Gamma \vdash (\tau_1[\alpha{:=}\tau_2]) : *$*.*

Lemma A.7. *If* $\Gamma \vdash \tau : \kappa$*, then* $\overline{\Gamma} \vdash [\![\tau]\!] : \kappa$*.*

Proof. By induction on the structure of τ.

Suppose τ is a type variable α. Then $(\alpha{:}\kappa) \in \Gamma$, and $(\alpha{:}\kappa) \in \overline{\Gamma}$, and $[\![\tau]\!] = \alpha$. Therefore, $\overline{\Gamma} \vdash [\![\tau]\!] : \kappa$ as required.

Suppose τ is an arrow type $\tau_1 \to \tau_2$. Then $\kappa = *$, and $\Gamma \vdash \tau_1 : *$, and $\Gamma \vdash \tau_2 : *$. By induction, $\overline{\Gamma} \vdash [\![\tau_1]\!] : *$, and $\overline{\Gamma} \vdash [\![\tau_2]\!] : *$, and $[\![\tau]\!] = \mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]$. By definition, we have that $(\mathsf{F}{:}* \to *) \in \overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash \mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!] : *$, as required.

Suppose τ is a quantified type $\forall\alpha{:}\kappa_1.\tau_1$. Then $\kappa = *$, and $\Gamma, \alpha{:}\kappa_1 \vdash \tau_1 : *$, and $[\![\tau]\!] = \forall\alpha{:}\kappa_1.\ \mathsf{F}\ [\![\tau_1]\!]$. By induction, $\overline{\Gamma, \alpha{:}\kappa_1} \vdash [\![\tau_1]\!] : *$. By definition, we have that $(\mathsf{F}{:}* \to *) \in \overline{\Gamma, \alpha{:}\kappa_1}$. Therefore, $\overline{\Gamma, \alpha{:}\kappa_1} \vdash \mathsf{F}\ [\![\tau_1]\!] : *$. By definition, we have that $\overline{\Gamma,\alpha{:}\kappa_1} = \overline{\Gamma},\alpha{:}\kappa_1$. Therefore, $\overline{\Gamma} \vdash (\forall\alpha{:}\kappa_1.\ \mathtt{F}\ [\![\tau_1]\!])\ :\ *$, as required.

Suppose τ is a λ-abstraction $(\lambda\alpha{:}\kappa_1.\tau_1)$. Then $\kappa = \kappa_1 \to \kappa_2$, and $\Gamma,\alpha{:}\kappa_1 \vdash \tau_1\ :\ \kappa_2$, and $[\![\tau]\!] = (\lambda\alpha{:}\kappa_1.[\![\tau_1]\!])$. By induction, $\overline{\Gamma,\alpha{:}\kappa_1} \vdash [\![\tau_1]\!]\ :\ \kappa_2$. By definition, we have that $\overline{\Gamma,\alpha{:}\kappa_1} = \overline{\Gamma},\alpha{:}\kappa_1$. Therefore, $\overline{\Gamma} \vdash (\lambda\alpha{:}\kappa_1.[\![\tau_1]\!])\ :\ \kappa_1 \to \kappa_2$, as required.

Suppose τ is an application $\tau_1\ \tau_2$. Then $\Gamma \vdash \tau_1\ :\ \kappa_1 \to \kappa$ and $\Gamma \vdash \tau_2\ :\ \kappa_1$, and $[\![\tau]\!] = ([\![\tau_1]\!]\ [\![\tau_2]\!])$. By induction, $\overline{\Gamma} \vdash [\![\tau_1]\!]\ :\ \kappa_1 \to \kappa$, and $\overline{\Gamma} \vdash [\![\tau_2]\!]\ :\ \kappa_1$. Therefore, $\overline{\Gamma} \vdash ([\![\tau_1]\!]\ [\![\tau_2]\!])\ :\ \kappa$, as required. □

Lemma A.8. *If $\Gamma \vdash \kappa$, then $\kappa \blacktriangleright \sigma$ for some σ.*

Proof. By induction on the structure of κ.

Suppose κ is $*$. Then $\sigma{=}(\forall\alpha{:}*.\alpha)$.

Suppose κ is $\kappa_1 \to \kappa_2$. By induction, $\kappa_2 \blacktriangleright \sigma_2$, and $\sigma{=}\lambda\alpha{:}\kappa_1.\sigma_2$. □

Lemma A.9. *If $\kappa \blacktriangleright \sigma$, then $\langle\rangle \vdash \sigma\ :\ \kappa$.*

Proof. By induction on the structure of κ.

Suppose $\kappa{=}*$. Then $\sigma{=}(\forall\alpha{:}*.\alpha)$, and $\langle\rangle \vdash \sigma\ :\ *$ as required.

Suppose $\kappa{=}\kappa_1 \to \kappa_2$. Then $\sigma{=}(\lambda\alpha{:}\kappa_1.\sigma_2)$ and $\kappa_2 \blacktriangleright \sigma_2$. By induction, $\langle\rangle \vdash \sigma_2\ :\ \kappa_2$. Therefore, $\langle\rangle \vdash (\lambda\alpha{:}\kappa_1.\sigma_2)\ :\ \kappa_1 \to \kappa_2$, as required. □

A.3 Section 6 Proofs

Lemma A.10. *If $\Gamma \vdash (\forall\alpha{:}\kappa.\tau)\ :\ *$, then $\overline{\Gamma} \vdash \mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])}\ :\ \mathtt{Strip\ F}\ [\![\forall\alpha{:}\kappa.\tau]\!]$.*

Proof. Suppose $\Gamma \vdash (\forall\alpha{:}\kappa.\tau)\ :\ *$. Then $[\![\forall\alpha{:}\kappa.\tau]\!] = (\forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!])$. We want to show that then $\mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])}$ has the type $(\mathtt{Strip\ F}\ [\![\forall\alpha{:}\kappa.\tau]\!])$.

Since $\Gamma \vdash (\forall\alpha{:}\kappa.\tau)\ :\ *$, Lemma A.16 states that $\Gamma,(\alpha{:}\kappa) \vdash \tau\ :\ *$. Therefore, $\Gamma \vdash (\lambda\alpha{:}\kappa.\tau)\ :\ \kappa \to *$. By Lemma A.7, $\overline{\Gamma} \vdash [\![\forall\alpha{:}\kappa.\tau]\!]\ :\ *$ and $\overline{\Gamma} \vdash [\![\lambda\alpha{:}\kappa.\tau]\!]\ :\ \kappa \to *$. By Lemma A.8, there exists a type σ such that $\kappa \rhd \sigma$. By Lemma A.9, $\langle\rangle \vdash \sigma\ :\ \kappa$. Therefore, $\Gamma \vdash \sigma\ :\ \kappa$.

We have that $\mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])} = \Lambda\alpha{:}*.\ \lambda\mathtt{f}{:}(\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha).\ \lambda\mathtt{x}{:}(\forall\gamma{:}\kappa.\mathtt{F}\ ([\![\lambda\alpha{:}\kappa.\tau]\!]\ \gamma)).\ \mathtt{f}\ ([\![\lambda\alpha{:}\kappa.\tau]\!]\ \sigma)\ (\mathtt{x}\ \sigma)$, so $\Gamma \vdash \mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])}\ :\ \forall\alpha{:}*.\ (\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha) \to (\forall\gamma{:}\kappa.\mathtt{F}\ ([\![\lambda\alpha{:}\kappa.\tau]\!]\ \gamma)) \to \alpha$. Also, $(\forall\gamma{:}\kappa.\mathtt{F}\ ([\![\lambda\alpha{:}\kappa.\tau]\!]\ \gamma)) \equiv (\forall\alpha{:}\kappa.\mathtt{F}\ ([\![\lambda\alpha{:}\kappa.\tau]\!]\ \alpha)) \equiv (\forall\alpha{:}\kappa.\mathtt{F}\ ((\lambda\alpha{:}\kappa.[\![\tau]\!])\ \alpha)) \equiv (\forall\alpha{:}\kappa.\mathtt{F}\ [\![\tau]\!]) = [\![\forall\alpha{:}\kappa.\tau]\!]$. Therefore, $\forall\alpha{:}*.\ (\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha) \to (\forall\gamma{:}\kappa.\mathtt{F}\ ((\lambda\alpha{:}\kappa.[\![\tau]\!])\ \gamma)) \to \alpha \equiv (\forall\alpha{:}*.\ (\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha) \to [\![\forall\alpha{:}\kappa.\tau]\!] \to \alpha) \equiv (\mathtt{Strip\ F}\ [\![\forall\alpha{:}\kappa.\tau]\!])$.

By the conversion typing rule, we have that $\Gamma \vdash \mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])}\ :\ \mathtt{Strip\ F}\ [\![\forall\alpha{:}\kappa.\tau]\!]$, as required. □

Lemma A.11 (Semantics of strip functions). *If $\kappa \blacktriangleright \sigma$, then $\mathtt{strip}_{(\mathtt{F},\kappa,\tau1)}\ \tau_2\ (\Lambda\alpha{:}*.\lambda\mathtt{x}{:}\tau_2.\mathtt{x})\ (\Lambda\beta{:}\kappa.\mathtt{e}) \longrightarrow^* \mathtt{e}[\beta{:=}\sigma]$.*

Proof. Suppose $\kappa \blacktriangleright \sigma$. Then

$$\begin{aligned}
&\mathtt{strip}_{(\mathtt{F},\kappa,\tau1)}\ \tau_2\ (\Lambda\alpha{:}*.\lambda\mathtt{x}{:}\tau_2.\mathtt{x})\ (\Lambda\beta{:}\kappa.\mathtt{e})\\
=\ &(\Lambda\alpha{:}*.\lambda\mathtt{f}{:}(\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha).\\
&\ \ \lambda\mathtt{x}{:}(\forall\gamma{:}\kappa.\mathtt{F}\ (\tau_1\ \gamma)).\ \mathtt{f}\ (\tau_1\ \sigma)\ (\mathtt{x}\ \sigma))\\
&\ \ \tau_2\ (\Lambda\alpha{:}*.\lambda\mathtt{x}{:}\tau_2.\mathtt{x})\ (\Lambda\beta{:}\kappa.\mathtt{e})\\
\longrightarrow^3\ &(\Lambda\alpha{:}*.\lambda\mathtt{x}{:}\tau_2.\mathtt{x})\ (\tau_1\ \sigma)\ ((\Lambda\beta{:}\kappa.\mathtt{e})\ \sigma)\\
\longrightarrow^2\ &((\Lambda\beta{:}\kappa.\mathtt{e})\ \sigma)\\
\longrightarrow\ &\mathtt{e}[\beta{:=}\sigma]
\end{aligned}$$

□

Lemma A.12 (Types of inst functions). *If $\Gamma \vdash (\forall\alpha{:}\kappa.\tau)\ :\ *$ and $\Gamma \vdash \sigma\ :\ \kappa$, then $\Gamma \vdash \mathtt{inst}_{(\forall\alpha:\kappa.\tau),\sigma}\ :\ (\forall\alpha{:}\kappa.\tau) \to (\tau[\alpha{:=}\sigma])$.*

Proof. Straightforward. □

Lemma A.13 (Semantics of inst functions). *If $\Gamma \vdash \mathtt{e}\ :\ (\forall\alpha{:}\kappa.\tau)$ and $\Gamma \vdash \sigma\ :\ \kappa$, then $\mathtt{inst}_{(\forall\alpha:\kappa.\tau),\sigma}\ \mathtt{e} \equiv_\beta \mathtt{e}\ \sigma$.*

Proof. Straightforward. □

Lemma A.14. *If $\Gamma \vdash \mathtt{e}\ :\ \tau$, then $\Gamma \vdash \tau\ :\ *$.*

Proof. Straightforward. □

Lemma A.15. *If $\Gamma,(\alpha{:}\kappa_1) \vdash \tau\ :\ \kappa_2$ and $\Gamma \vdash \sigma\ :\ \kappa_1$, then $\Gamma \vdash (\tau[\alpha{:=}\sigma])\ :\ \kappa_2$.*

Proof. Standard. □

Lemma A.16. *1. If $\Gamma \vdash \tau_1 \to \tau_2\ :\ *$, then $\Gamma \vdash \tau_1\ :\ *$ and $\Gamma \vdash \tau_2\ :\ *$.*
2. If $\Gamma \vdash (\forall\alpha{:}\kappa.\tau)\ :\ *$, then $\Gamma,(\alpha{:}\kappa) \vdash \tau\ :\ *$.

Proof. Standard. □

Lemma A.17. *If $\Gamma,(\alpha{:}\kappa_1) \vdash \tau\ :\ \kappa_2$ and $\Gamma \vdash \sigma\ :\ \kappa_1$, then $\Gamma \vdash (\tau[\alpha{:=}\sigma])\ :\ \kappa_2$.*

Proof. Standard. □

Definition A.5 (Environment for Pre-quotations).

$$\begin{array}{ll}
\overline{\langle\rangle} & = (\mathtt{F}{:}* \to *),\\
 & \quad (\mathtt{abs{:}Abs\ F}),(\mathtt{app{:}App\ F}),\\
 & \quad (\mathtt{tabs{:}TAbs\ F}),(\mathtt{tapp{:}TApp\ F})\\
\overline{\Gamma,\mathtt{x}{:}\tau} & = \overline{\Gamma},\mathtt{x}{:}\mathtt{F}\ [\![\tau]\!]\\
\overline{\Gamma,\alpha{:}\kappa} & = \overline{\Gamma},\alpha{:}\kappa
\end{array}$$

Theorem A.1. *If $\Gamma \vdash \mathtt{e}\ :\ \tau \rhd \mathtt{q}$, then $\overline{\Gamma} \vdash \mathtt{q}\ :\ \mathtt{F}\ [\![\tau]\!]$.*

Proof. By induction on the height of the derivation $\Gamma \vdash \mathtt{e}\ :\ \tau \rhd \mathtt{q}$.

Suppose $\mathtt{e}$ is a variable $\mathtt{x}$. Then $(\mathtt{x}{:}\tau) \in \Gamma$, and $\mathtt{q} = \mathtt{x}$. Since $(\mathtt{x}{:}\tau) \in \Gamma$, $(\mathtt{x}{:}\mathtt{F}\ [\![\tau]\!]) \in \overline{\Gamma}$. Therefore, $\Gamma \vdash \mathtt{q}\ :\ \mathtt{F}\ [\![\tau]\!]$, as required.

Suppose $\mathtt{e}$ is a λ-abstraction $(\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1)$. Then $\tau = \tau_1 \to \tau_2$, and $[\![\tau]\!] = \mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$, and $\Gamma,\mathtt{x}{:}\tau_1 \vdash \mathtt{e}_1\ :\ \tau_2 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$. We want to show that $\overline{\Gamma} \vdash \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)\ :\ \mathtt{F}\ (\mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!])$. By the definition of $\overline{\Gamma}$, we have $(\mathtt{abs{:}Abs\ F}) \in \overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash \mathtt{abs}\ :\ (\forall\alpha{:}*.\ \forall\beta{:}*.\ (\mathtt{F}\ \alpha \to \mathtt{F}\ \beta) \to \mathtt{F}\ (\mathtt{F}\ \alpha \to \mathtt{F}\ \beta))$. By Lemma A.14, $\Gamma \vdash \tau_1 \to \tau_2\ :\ *$. By Lemma A.16, $\Gamma \vdash \tau_1\ :\ *$ and $\Gamma \vdash \tau_2\ :\ *$. By Lemma A.7, $\overline{\Gamma} \vdash [\![\tau_1]\!]\ :\ *$ and $\overline{\Gamma} \vdash [\![\tau_2]\!]\ :\ *$. By induction, $\overline{\Gamma,(\mathtt{x}{:}\tau_1)} \vdash \mathtt{q}_1\ :\ \mathtt{F}\ [\![\tau_2]\!]$, so $\overline{\Gamma} \vdash (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)\ :\ \mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!]$. Therefore, $\overline{\Gamma} \vdash \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)\ :\ \mathtt{F}\ (\mathtt{F}\ [\![\tau_1]\!] \to \mathtt{F}\ [\![\tau_2]\!])$, as required.

Suppose $\mathtt{e}$ is an application $\mathtt{e}_1\ \mathtt{e}_2$. Then $\Gamma \vdash \mathtt{e}_1\ :\ \tau_2 \to \tau \rhd \mathtt{q}_1$, and $\Gamma \vdash \mathtt{e}_2\ :\ \tau_2 \rhd \mathtt{q}_2$, and $\mathtt{q} = \mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2$. We want to show $\overline{\Gamma} \vdash \mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2\ :\ \mathtt{F}\ [\![\tau]\!]$. By the definition of $\overline{\Gamma}$, we have $(\mathtt{app{:}App\ F}) \in \overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash \mathtt{app}\ :\ (\forall\alpha{:}*.\ \forall\beta{:}*.\ \mathtt{F}\ (\mathtt{F}\ \alpha \to \mathtt{F}\ \beta) \to \mathtt{F}\ \alpha \to \mathtt{F}\ \beta)$. By Lemma A.14, $\Gamma \vdash \tau_2\ :\ *$ and $\Gamma \vdash \tau\ :\ *$. By Lemma A.7, $\overline{\Gamma} \vdash [\![\tau_2]\!]\ :\ *$ and $\overline{\Gamma} \vdash [\![\tau]\!]\ :\ *$. By induction, $\overline{\Gamma} \vdash \mathtt{q}_1\ :\ \mathtt{F}\ [\![\tau_2 \to \tau]\!]$, and $\overline{\Gamma} \vdash \mathtt{q}_2\ :\ \mathtt{F}\ [\![\tau_2]\!]$. Since $[\![\tau_2 \to \tau]\!] = \mathtt{F}\ [\![\tau_2]\!]$

$\to \mathtt{F}\ [\![\tau]\!]$, $\overline{\Gamma} \vdash \mathtt{q}_1 : \mathtt{F}\ (\mathtt{F}\ [\![\tau_2]\!] \to \mathtt{F}\ [\![\tau]\!])$. Therefore $\overline{\Gamma} \vdash \mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2 : \mathtt{F}\ [\![\tau]\!]$, as required.

Suppose $\mathtt{e}$ is a type abstraction $\Lambda\alpha{:}\kappa.\mathtt{e}_1$. Then $\tau = \forall\alpha{:}\kappa.\tau_1$, and $[\![\tau]\!] = (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!])$, and $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and

$$\begin{aligned}\mathtt{q} = \mathtt{tabs}\ & [\![\forall\alpha{:}\kappa.\tau_1]\!] \\ & \mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau 1]\!])} \\ & (\Lambda\alpha{:}\kappa.\mathtt{q})\end{aligned}$$

We want to show $\overline{\Gamma} \vdash \mathtt{q} : \mathtt{F}\ (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!])$. By the definition of $\overline{\Gamma}$, we have $(\mathtt{tabs{:}TAbs\ F}) \in \overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash \mathtt{tabs} : (\forall\alpha{:}*.\ \mathtt{Strip\ F}\ \alpha \to \alpha \to \mathtt{F}\ \alpha)$. By Lemma A.14, $\Gamma \vdash (\forall\alpha{:}\kappa.\tau_1) : *$. By Lemma A.7, we have that $\overline{\Gamma} \vdash [\![\forall\alpha{:}\kappa.\tau_1]\!] : *$. Then $\overline{\Gamma} \vdash \mathtt{tabs}\ [\![\forall\alpha{:}\kappa.\tau_1]\!] : \mathtt{Strip\ F}\ [\![\forall\alpha{:}\kappa.\tau_1]\!] \to [\![\forall\alpha{:}\kappa.\ \tau_1]\!] \to \mathtt{F}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]$, so it suffices to show that (1) $\overline{\Gamma} \vdash \mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:\kappa.\tau 1]\!])} : \mathtt{Strip\ F}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]$, and that (2) $\overline{\Gamma} \vdash (\Lambda\alpha{:}\kappa.\mathtt{q}) : [\![\forall\alpha{:}\kappa.\ \tau_1]\!]$.

(1) Is given by Lemma A.10.

(2) By induction, $\overline{\Gamma,\alpha{:}\kappa} \vdash \mathtt{q}_1 : \mathtt{F}\ [\![\tau_1]\!]$, so $\overline{\Gamma} \vdash (\Lambda\alpha{:}\kappa.\mathtt{q}_1) : (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!])$. By definition, $(\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]) = [\![\forall\alpha{:}\kappa.\tau_1]\!]$, as required.

Suppose $\mathtt{e}$ is a type application $\mathtt{e}_1\ \sigma$. Then $\tau{=}(\tau_1[\alpha{:=}\sigma])$, and $\Gamma \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathtt{q}_1$, and $\Gamma \vdash \sigma : \kappa$, and

$$\begin{aligned}\mathtt{q} = \mathtt{tapp}\ & (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]) \\ & \mathtt{q}_1\ [\![\tau_1[\alpha{:=}\sigma]]\!] \\ & \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],\sigma)}\end{aligned}$$

By the definition of $\overline{\Gamma}$, we have $(\mathtt{tapp{:}TApp\ F}) \in \overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash \mathtt{tapp} : (\forall\alpha{:}*.\ \mathtt{F}\ \alpha \to \forall\beta{:}*.\ (\alpha \to \mathtt{F}\ \beta) \to \mathtt{F}\ \beta)$. Then $\overline{\Gamma} \vdash \mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_1]\!] : \mathtt{F}\ [\![\forall\alpha{:}\kappa.\tau_1]\!] \to \forall\beta{:}*.\ ([\![\forall\alpha{:}\kappa.\tau_1]\!] \to \mathtt{F}\ \beta) \to \mathtt{F}\ \beta$. We want to show: (1) $\overline{\Gamma} \vdash \mathtt{q}_1 : \mathtt{F}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]$, and (2) $\overline{\Gamma} \vdash [\![\tau_1[\alpha{:=}\sigma]]\!] : *$, and (3) $\overline{\Gamma} \vdash \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],\sigma)} : [\![\forall\alpha{:}\kappa.\tau_1]\!] \to \mathtt{F}\ [\![\tau_1[\alpha{:=}\sigma]]\!]$.
(1) is by the induction hypothesis.
For (2), we have $\Gamma \vdash (\forall\alpha{:}\kappa.\tau_1) : *$, and by Lemma A.16 $\Gamma,(\alpha{:}\kappa) \vdash \tau_1 : *$. Since $\Gamma \vdash \sigma : \kappa$, Lemma A.15 states that $\Gamma \vdash (\tau_1[\alpha{:=}\sigma]) : *$. By Lemma A.7, $\overline{\Gamma} \vdash [\![\tau_1[\alpha{:=}\sigma]]\!] : *$, as required.
(3) is by Theorem A.12. □

Theorem 6.1. *If $\langle\rangle \vdash \mathtt{e} : \tau$, then $\langle\rangle \vdash \overline{\mathtt{e}} : \mathtt{Exp}\ \overline{\tau}$.*

Proof. Suppose $\langle\rangle \vdash \mathtt{e} : \tau$. Let $\mathtt{q}$ be such that $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$. By Lemma A.1, we have $\overline{\langle\rangle} \vdash \mathtt{q} : \mathtt{F}\ [\![\tau]\!]$. By definition,

$$\begin{aligned}\overline{\langle\rangle} = \ & (\mathtt{F}{:}* \to *), \\ & (\mathtt{abs{:}Abs\ F}),(\mathtt{app{:}App\ F}), \\ & (\mathtt{tabs{:}TAbs\ F}),(\mathtt{tapp{:}TApp\ F})\end{aligned}$$

The result follows from the definitions of $\overline{\mathtt{e}}$ and $\mathtt{Exp}$. □

The following Lemma states that strip functions are always normal forms. Recall that normal forms can have redexes in the types.

Lemma A.18. *For all kinds κ and types $\mathtt{F}$, and τ, $\mathtt{strip}_{(\mathtt{F},\kappa,\tau)}$ is normal.*

Proof. We have that $\mathtt{strip}_{(\mathtt{F},\kappa,\tau)} = \Lambda\alpha{:}*.\ \lambda\mathtt{f}{:}(\forall\beta{:}*.\ \mathtt{F}\ \beta \to \alpha).\ \lambda\mathtt{x}{:}(\forall\gamma{:}\kappa.\mathtt{F}\ (\tau\ \gamma)).\ \mathtt{f}\ (\tau\ \sigma)\ (\mathtt{x}\ \sigma)$. Since $\mathtt{f}$ and $\mathtt{x}$ are variables, $\mathtt{strip}_{(\mathtt{F},\kappa,\tau)}$ is normal. □

Lemma A.19. *For any types τ and σ, $\mathtt{inst}_{(\tau,\sigma)}$ is normal.*

Proof. $\mathtt{inst}_{(\tau,\sigma)} = (\lambda\mathtt{x}{:}\tau.\ \mathtt{x}\ \sigma)$. Since $\mathtt{x}$ is a variable, $\mathtt{inst}_{(\tau,\sigma)}$ is normal. □

Lemma A.20. *If $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, then $\mathtt{q}$ is β-normal.*

Proof. By induction on the structure of $\mathtt{e}$.

Suppose $\mathtt{e}$ is a variable $\mathtt{x}$. Then $\mathtt{q} = \mathtt{x}$, which is normal.

Suppose $\mathtt{e}$ is a λ-abstraction $\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1$. Then $\Gamma,\mathtt{x}{:}\tau_1 \vdash \mathtt{e}_1 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$. By the induction hypothesis, $\mathtt{q}_1$ is normal. Then $(\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$ is normal, and since $\mathtt{abs}$ is a variable, it is normal and neutral. Therefore, $\mathtt{q}$ is normal.

Suppose $\mathtt{e}$ is an application $\mathtt{e}_1\ \mathtt{e}_2$. Then $\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1$ and $\Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2$, and $\mathtt{q} = \mathtt{app}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ \mathtt{q}_1\ \mathtt{q}_2$. By the induction hypothesis, $\mathtt{q}_1$ and $\mathtt{q}_2$ are normal. Since $\mathtt{app}$ is a variable, it is normal and neutral. Therefore, $\mathtt{q}$ is normal.

Suppose $\mathtt{e}$ is a Λ-abstraction $\Lambda\alpha{:}\kappa.\mathtt{e}_1$. Then $\tau = (\forall\alpha{:}\kappa.\tau_1)$, and $\Gamma \vdash \kappa \blacktriangleright \sigma$, and $\Gamma,\ \alpha{:}\kappa \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and

$$\begin{aligned}\mathtt{q} = \mathtt{tabs}\ & (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!]) \\ & \mathtt{strip}_{(\mathtt{F},\kappa,\lambda\alpha:\kappa.[\![\tau]\!])} \\ & (\Lambda\alpha{:}\kappa.\mathtt{q}_1)\end{aligned}$$

Since $\mathtt{tabs}$ is a variable, it is normal and neutral. By Lemma A.18, $\mathtt{strip}_{(\mathtt{F},\kappa,\lambda\alpha:\kappa.[\![\tau]\!])}$ is normal. By the induction hypothesis, $\mathtt{q}_1$ is normal. Therefore, $(\Lambda\alpha{:}\kappa.\mathtt{q}_1)$ is normal. Therefore, $\mathtt{q}$ is normal.

Suppose $\mathtt{e}$ is a type application $\mathtt{e}_1\ \sigma$. Then $\tau = \tau_1[\alpha{:=}\sigma]$, and $\Gamma \vdash \mathtt{e} : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathtt{q}_1$, and

$$\begin{aligned}\mathtt{q} = \mathtt{tapp}\ & (\forall\alpha{:}\kappa.\ \mathtt{F}\ [\![\tau_1]\!])\ \mathtt{q}_1\ [\![\tau_1[\alpha{:=}\sigma]]\!] \\ & \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],\sigma)}\end{aligned}$$

Since $\mathtt{tapp}$ is a variable, it is normal and neutral. By induction q_1 is normal. By Lemma A.19, $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],\sigma)}$ is normal. Therefore, $\mathtt{q}$ is normal. □

Theorem 6.2. *If $\langle\rangle \vdash \mathtt{e} : \tau$, then $\overline{\mathtt{e}}$ is β-normal.*

Proof. Suppose $\langle\rangle \vdash \mathtt{e} : \tau$. Then $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, and

$$\begin{aligned}\overline{\mathtt{e}} = \ & \Lambda\mathtt{F}{:}* \to *. \\ & \lambda\mathtt{abs}\ {:}\mathtt{Abs\ F}.\quad \lambda\mathtt{app}\ {:}\mathtt{App\ F}. \\ & \lambda\mathtt{tabs}{:}\mathtt{TAbs\ F}.\ \lambda\mathtt{tapp}{:}\mathtt{TApp\ F}. \\ & \mathtt{q}\end{aligned}$$

By Lemma A.20, $\mathtt{q}$ is normal. Therefore, $\overline{\mathtt{e}}$ is normal. □

Definition A.6. *We define a similarity relation between environments that reflects that each environment assigns term variables to equivalent types as follows:*

$$\frac{}{\langle\rangle \sim \langle\rangle}$$

$$\frac{\Gamma \sim \Gamma' \qquad \tau \equiv \tau'}{\Gamma,(\mathtt{x}{:}\tau) \sim \Gamma',(\mathtt{x}{:}\tau')}$$

$$\frac{\Gamma \sim \Gamma'}{\Gamma,(\alpha{:}\kappa) \sim \Gamma',(\alpha{:}\kappa)}$$

Lemma A.21. *If $\Gamma \vdash \mathtt{e} : \tau$ and $\Gamma' \vdash \mathtt{e}' : \tau'$, and $\Gamma \sim \Gamma'$ and $\mathtt{e} \sim \mathtt{e}'$, then $\tau \equiv \tau'$.*

Proof. By induction on the derivation $\Gamma \vdash \mathtt{e} : \tau$.

Suppose $\Gamma \vdash \mathtt{e} : \tau$ is by the first rule. Then $\mathtt{e}$ is a variable $\mathtt{x}$, and so $\mathtt{e}'$ is also $\mathtt{x}$. We have $(\mathtt{x}{:}\tau) \in \Gamma$ and $(\mathtt{x}{:}\tau') \in \Gamma'$, and $\tau \equiv \tau'$ follows from $\Gamma \sim \Gamma'$.

Suppose $\mathtt{e}$ is by the second rule. Then $\mathtt{e} = (\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1)$, and $\tau = \tau_1 \to \tau_2$, $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2$. Since $\mathtt{e} \sim \mathtt{e}'$, we have that $\mathtt{e}' = (\lambda\mathtt{x}{:}\tau_1'.\mathtt{e}_1')$, and $\tau_1 \equiv \tau_1'$ and $\mathtt{e}_1 \sim \mathtt{e}_1'$. Further $\tau' = \tau_1' \to \tau_2'$ and $\Gamma,(\mathtt{x}{:}\tau_1') \vdash \mathtt{e}_1' : \tau_2'$. Since $\Gamma \sim \Gamma'$, we have that $\Gamma,(\mathtt{x}{:}\tau_1) \sim \Gamma',(\mathtt{x}{:}\tau_1')$. Therefore, by induction, $\tau_2 \equiv \tau_2'$. Therefore, $\tau \equiv \tau'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau$ is by the third rule. Then $\mathtt{e} = (\mathtt{e}_1\ \mathtt{e}_2)$, and $\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau$. Since $\mathtt{e} \sim \mathtt{e}'$, we have that $\mathtt{e}' = (\mathtt{e}_1'\ \mathtt{e}_2')$, and $\mathtt{e}_1 \sim \mathtt{e}_1'$. Also, $\Gamma' \vdash \mathtt{e}_1' : \tau_2' \to \tau'$, so by induction we have that $(\tau_2 \to \tau) \equiv (\tau_2' \to \tau')$. Therefore, $\tau \equiv \tau'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau$ is by the fourth rule. Then $\mathtt{e} = (\Lambda\alpha{:}\kappa.\mathtt{e}_1)$, and $\tau = (\forall\alpha{:}\kappa.\tau_1)$, and $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1$. Since $\mathtt{e} \sim \mathtt{e}'$, $\mathtt{e}' = (\Lambda\alpha{:}\kappa.\mathtt{e}_1')$. Therefore, $\tau' = (\forall\alpha{:}\kappa.\tau_1')$ and $\Gamma',(\alpha{:}\kappa) \vdash \mathtt{e}_1' : \tau_1'$. Since $\Gamma \sim \Gamma'$, we have that $\Gamma,(\alpha{:}\kappa) \sim \Gamma',(\alpha{:}\kappa)$. By induction, $\tau_1 \equiv \tau_1'$. Therefore, $(\forall\alpha{:}\kappa.\tau_1) \equiv (\forall\alpha{:}\kappa.\tau_1')$, so $\tau \equiv \tau'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau$ is by the fifth rule. Then $\mathtt{e} = (\mathtt{e}_1\ \sigma)$, and $\Gamma \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau_1)$, and $\tau = (\tau_1[\alpha{:=}\sigma])$. Since $\mathtt{e} \sim \mathtt{e}'$, we have that $\mathtt{e}' = (\mathtt{e}_1'\ \sigma')$, and $\sigma \equiv \sigma'$. Since $\Gamma' \vdash \mathtt{e}' : \tau'$, we have that $\Gamma' \vdash \mathtt{e}_1' : (\forall\alpha{:}\kappa.\tau_1')$ and $\tau' = (\tau_1'[\alpha{:=}\sigma'])$. By induction, $(\forall\alpha{:}\kappa.\tau_1) \equiv (\forall\alpha{:}\kappa.\tau_1')$. Therefore, $\tau_1 \equiv \tau_1'$, so $(\lambda\alpha{:}\kappa.\tau_1) \equiv (\lambda\alpha{:}\kappa.\tau_1')$, so $((\lambda\alpha{:}\kappa.\tau_1)\ \sigma) \equiv ((\lambda\alpha{:}\kappa.\tau_1')\ \sigma')$, so $(\tau_1[\alpha{:=}\sigma]) \equiv (\tau_1'[\alpha{:=}\sigma'])$, so $\tau \equiv \tau'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau$ is by the sixth rule. Then $\Gamma \vdash \mathtt{e} : \sigma$ and $\sigma \equiv \tau$. By induction, $\sigma \equiv \tau'$. Therefore, $\tau \equiv \tau'$. ☐

Lemma A.22. *If $\tau \equiv \tau'$, then $\mathtt{strip}_{(\mathsf{F},\kappa,[\![\lambda\alpha:\kappa.\tau]\!])} \sim \mathtt{strip}_{(\mathsf{F},\kappa,[\![\lambda\alpha:\kappa.\tau']\!])}$.*

Proof. Straightforward. ☐

Lemma A.23. *If $\tau \equiv \tau'$ and $\sigma \equiv \sigma'$, then $\mathtt{inst}_{(\tau,\sigma)} \sim \mathtt{inst}_{(\tau',\sigma')}$.*

Proof. Straightforward. ☐

Lemma A.24. *If $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, and $\Gamma' \vdash \mathtt{e}' : \tau' \rhd \mathtt{q}'$, and $\Gamma \sim \Gamma'$, and $\mathtt{e} \sim \mathtt{e}'$, then $\mathtt{q} \sim \mathtt{q}'$.*

Proof. By induction on the derivation $\mathtt{e} \sim \mathtt{e}'$.

Suppose $\mathtt{e} \sim \mathtt{e}'$ is $\mathtt{x} \sim \mathtt{x}$. Then $\mathtt{e} = \mathtt{e}' = \mathtt{x}$, so $\mathtt{q} = \mathtt{q}' = \mathtt{x}$. Therefore, $\mathtt{q} \sim \mathtt{q}'$.

Suppose $\mathtt{e} \sim \mathtt{e}'$ is $(\lambda\mathtt{x}{:}\tau.\mathtt{e}_1) \sim (\lambda\mathtt{x}{:}\tau'.\mathtt{e}_1{}')$. Then $\mathtt{e} = (\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1)$ and $\tau = \tau_1 \to \tau_2$, and $\mathtt{e}' = (\lambda\mathtt{x}{:}\tau_1'.\mathtt{e}_1')$ and $\tau' = \tau_1' \to \tau_2'$, and $\tau_1 \equiv \tau_1'$ and $\mathtt{e}_1 \sim \mathtt{e}_1'$. Also, $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$, and similarly, $\Gamma',(\mathtt{x}{:}\tau_1{}') \vdash \mathtt{e}_1{}' : \tau_2{}' \rhd \mathtt{q}_1{}'$, and $\mathtt{q}' = \mathtt{abs}\ [\![\tau_1{}']\!]\ [\![\tau_2{}']\!]\ (\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1{}']\!].\ \mathtt{q}_1{}')$. Since $\Gamma \sim \Gamma'$ and $\tau_1 \sim \tau_1'$, we have that $\Gamma,(\mathtt{x}{:}\tau_1) \sim \Gamma',(\mathtt{x}{:}\tau_1{}')$. Therefore, the induction hypothesis states that $\mathtt{q}_1 \sim \mathtt{q}_1{}'$, and it is easily checked that $\mathtt{q} \sim \mathtt{q}'$.

Suppose $\mathtt{e} \sim \mathtt{e}'$ is $(\mathtt{e}_1\ \mathtt{e}_2) \sim (\mathtt{e}_1{}'\ \mathtt{e}_2{}')$. Then $\mathtt{e} = (\mathtt{e}_1\ \mathtt{e}_2)$, and $\mathtt{e}' = (\mathtt{e}_1{}'\ \mathtt{e}_2{}')$, and $\mathtt{e}_1 \sim \mathtt{e}_1{}'$, and $\mathtt{e}_2 \sim \mathtt{e}_2{}'$. Therefore, $\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1$, and $\Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2$, and $\mathtt{q} = \mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2$. Similarly, $\Gamma' \vdash \mathtt{e}_1{}' : \tau_2{}' \to \tau' \rhd \mathtt{q}_1{}'$, and $\Gamma' \vdash \mathtt{e}_2{}' : \tau_2{}' \rhd \mathtt{q}_2{}'$, and $\mathtt{q}' = \mathtt{app}\ [\![\tau_2{}']\!]\ [\![\tau']\!]\ \mathtt{q}_1{}'\ \mathtt{q}_2{}'$. By induction $\mathtt{q}_1 \sim \mathtt{q}_1{}'$, and $\mathtt{q}_2 \sim \mathtt{q}_2{}'$, and it is easily checked that $\mathtt{q} \sim \mathtt{q}'$.

Suppose $\mathtt{e} \sim \mathtt{e}'$ is $(\Lambda\alpha{:}\kappa.\mathtt{e}_1) \sim (\Lambda\alpha{:}\kappa.\mathtt{e}_1{}')$. Then $\mathtt{e} = (\Lambda\alpha{:}\kappa.\mathtt{e}_1)$, and $\mathtt{e}' = (\Lambda\alpha{:}\kappa.\mathtt{e}_1')$, and $\mathtt{e}_1 \sim \mathtt{e}_1'$. Therefore, $\tau = (\forall\alpha{:}\kappa.\tau_1)$, $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{tabs}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{strip}_{(\mathsf{F},\kappa,\lambda\alpha:\kappa.[\![\tau_1]\!])}\ (\Lambda\alpha{:}\kappa.\mathtt{q})$. Similarly, $\tau' = (\forall\alpha{:}\kappa.\tau_1')$, $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1' : \tau_1' \rhd \mathtt{q}_1'$, and $\mathtt{q} = \mathtt{tabs}\ [\![\forall\alpha{:}\kappa.\tau_1']\!]\ \mathtt{strip}_{(\mathsf{F},\kappa,\lambda\alpha:\kappa.[\![\tau_1']\!])}\ (\Lambda\alpha{:}\kappa.\mathtt{q}_1')$. Since $\Gamma \sim \Gamma'$, we have that $\Gamma,(\alpha{:}\kappa) \sim \Gamma',(\alpha{:}\kappa)$. Therefore, by induction we have that $\mathtt{q}_1 \sim \mathtt{q}_1'$. By Lemma A.21 we have that $\tau_1 \sim \tau_1'$, so by Lemma A.22 $\mathtt{strip}_{(\mathsf{F},\kappa,\lambda\alpha:\kappa.[\![\tau_1]\!])} \sim \mathtt{strip}_{(\mathsf{F},\kappa,\lambda\alpha:\kappa.[\![\tau_1']\!])}$. Therefore, it is easily checked that $\mathtt{q} \sim \mathtt{q}'$.

Suppose $\mathtt{e} \sim \mathtt{e}'$ is $(\mathtt{e}_1\ \tau_1) \sim (\mathtt{e}_1{}'\ \tau_1{}')$. Then $\mathtt{e} = (\mathtt{e}_1\ \tau_1)$ and $\mathtt{e}' = (\mathtt{e}_1'\ \tau_1')$, and $\mathtt{e} \sim \mathtt{e}'$ and $\tau_1 \equiv \tau_1'$. Therefore, $\Gamma \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau_2) \rhd \mathtt{q}_1$ and $\Gamma \vdash \tau_1 : \kappa$ and $\tau = \tau_2[\alpha{:=}\tau_1]$ and $\mathtt{q} = (\mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_2]\!]\ \mathtt{q}_1\ [\![\tau_2[\alpha{:=}\tau_1]]\!]\ \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau_2]\!],[\![\tau_1]\!])})$. Similarly, $\Gamma \vdash \mathtt{e}_1' : (\forall\alpha{:}\kappa.\tau_2') \rhd \mathtt{q}_1'$ and $\Gamma \vdash \tau_1' : \kappa$ and $\tau = \tau_2'[\alpha{:=}\tau_1']$ and $\mathtt{q} = \mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_2']\!]\ \mathtt{q}_1'\ [\![\tau_2'[\alpha{:=}\tau_1']]\!]\ \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau_2']\!],[\![\tau_1']\!])}$. By induction, we have that $\mathtt{q}_1 \sim \mathtt{q}_1'$. By Lemma A.21, we have that $(\forall\alpha{:}\kappa.\tau_2) \equiv (\forall\alpha{:}\kappa.\tau_2')$. By Theorem 5.3, $[\![(\forall\alpha{:}\kappa.\tau_2)]\!] \equiv [\![(\forall\alpha{:}\kappa.\tau_2')]\!]$. Therefore, by Lemma A.23, $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau_2]\!],[\![\tau_1]\!])} \sim \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau_2']\!],[\![\tau_1']\!])}$. It is easily checked that $\mathtt{q} \sim \mathtt{q}'$. ☐

Lemma A.25. $\mathtt{inst}_{(\tau 1,\sigma 1)} \sim \mathtt{inst}_{(\tau 2,\sigma 2)}$*, then $\tau_1 \equiv \tau_2$ and $\sigma_1 \equiv \sigma_2$.*

Proof. We have that $\mathtt{inst}_{(\tau 1,\sigma 1)} = (\lambda\mathtt{x}{:}\tau_1.\ \mathtt{x}\ \sigma_1)$, and $\mathtt{inst}_{(\tau 2,\sigma 2)} = (\lambda\mathtt{x}{:}\tau_2.\ \mathtt{x}\ \sigma_2)$. Therefore, $(\lambda\mathtt{x}{:}\tau_1.\ \mathtt{x}\ \sigma_1) \sim (\lambda\mathtt{x}{:}\tau_2.\ \mathtt{x}\ \sigma_2)$, so $\tau_1 \equiv \tau_2$ and $\sigma_1 \equiv \sigma_2$. ☐

Lemma A.26. *If $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, and $\Gamma' \vdash \mathtt{e}' : \tau' \rhd \mathtt{q}'$, and $\Gamma \sim \Gamma'$, and $\mathtt{q} \sim \mathtt{q}'$, then $\mathtt{e} \sim \mathtt{e}'$.*

Proof. By induction on the derivation $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$.

Suppose $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$ is by the first rule. Then $\mathtt{q} = \mathtt{e} = \mathtt{x}$, so since $\mathtt{q} \sim \mathtt{q}'$, we have $= \mathtt{q}' = \mathtt{e}' = \mathtt{x}$. Therefore, $\mathtt{e} \sim \mathtt{e}'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$ is by the second rule. Then $\mathtt{e}{=}(\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1)$, and $\tau{=}\tau_1 \to \tau_2$, and $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$. Since $\mathtt{q} \sim \mathtt{q}'$, we have that $\mathtt{q}' = \mathtt{abs}\ \sigma_1\ \sigma_2\ \mathtt{f}$, and $[\![\tau_1]\!] \equiv \sigma_1$, and $[\![\tau_2]\!] \equiv \sigma_2$ and $(\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1]\!].\ \mathtt{q}_1) \sim \mathtt{f}$. Since $\Gamma' \vdash \mathtt{e}' : \tau' : \mathtt{q}'$, it must be the case that $\mathtt{e}' = (\lambda\mathtt{x}{:}\tau_1'.\mathtt{e}_1')$, and $\tau' = \tau_1' \to \tau_2'$, and $\Gamma',(\mathtt{x}{:}\tau_1') \vdash \mathtt{e}_1' : \tau_2' \rhd \mathtt{q}_1'$, and $\mathtt{q}' = \mathtt{abs}\ [\![\tau_1']\!]\ [\![\tau_2']\!]\ (\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1']\!].\ \mathtt{q}_1')$. So we have that $[\![\tau_1]\!] \equiv [\![\tau_1']\!]$, and $[\![\tau_2]\!] \equiv [\![\tau_2']\!]$, and $(\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1]\!].\ \mathtt{q}_1) \sim (\lambda\mathtt{x}{:}\mathsf{F}\ [\![\tau_1']\!].\ \mathtt{q}_1')$. Therefore $\mathtt{q}_1 \sim \mathtt{q}_1'$. By Theorem 5.3, we have that $\tau_1 \equiv \tau_2$ and $\tau_1' \equiv \tau_2'$. Therefore, $(\Gamma,(\mathtt{x}{:}\tau_1)) \sim (\Gamma',(\mathtt{x}{:}\tau_1'))$. By induction, we have that $\mathtt{e}_1 \sim \mathtt{e}_1'$. Therefore, $\mathtt{e} \sim \mathtt{e}'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$ is by the third rule. Then $\mathtt{e} = \mathtt{e}_1\ \mathtt{e}_2$, and $\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1$, and $\Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2$, and $\mathtt{q} = \mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2$. Since $\mathtt{q} \sim \mathtt{q}'$, we have that $\mathtt{q}' = \mathtt{app}\ \sigma_1\ \sigma_2\ \mathtt{a}_1\ \mathtt{a}_2$, and $[\![\tau_2]\!] \equiv \sigma_1$, and $[\![\tau]\!] \equiv \sigma_2$ and $\mathtt{q}_1 \sim \mathtt{a}_1$ and $\mathtt{q}_2 \sim \mathtt{a}_2$. Since $\Gamma' \vdash \mathtt{e}' : \tau' \rhd \mathtt{q}'$, it must be the case that $\mathtt{e}' = \mathtt{e}_1'\ \mathtt{e}_2'$, and $\Gamma' \vdash \mathtt{e}_1' : \tau_2' \to \tau' \rhd \mathtt{q}_1'$, and $\Gamma' \vdash \mathtt{e}_2' : \tau_2' \rhd \mathtt{q}_2'$, and $\mathtt{q}' = \mathtt{app}\ [\![\tau_2']\!]\ [\![\tau']\!]\ \mathtt{q}_1'\ \mathtt{q}_2'$. So $[\![\tau_2]\!] \equiv [\![\tau_2']\!]$, and $[\![\tau]\!] \equiv [\![\tau']\!]$, and $\mathtt{q}_1 \sim \mathtt{q}_1'$ and $\mathtt{q}_2 \sim \mathtt{q}_2'$. By induction, we have that $\mathtt{e}_1 \sim \mathtt{e}_1'$ and $\mathtt{e}_2 \sim \mathtt{e}_2'$. Therefore, $\mathtt{e} \sim \mathtt{e}'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$ is by the fourth rule. Then $\mathtt{e} = (\Lambda\alpha{:}\kappa.\mathtt{e}_1)$ and $\tau = (\forall\alpha{:}\kappa.\tau_1)$ and $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{tabs}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{strip}_{(\mathsf{F},\kappa,[\![\lambda\alpha:\kappa.\tau 1]\!])}\ (\Lambda\alpha{:}\kappa.\mathtt{q}_1)$. From now on we abbreviate $\mathtt{strip}_{(\mathsf{F},\kappa,[\![\lambda\alpha:\kappa.\tau 1]\!])}$ as $\mathtt{strip}$. Since $\mathtt{q} \sim \mathtt{q}'$, we have that $\mathtt{q}' = \mathtt{tabs}\ \sigma\ \mathtt{a}_1\ \mathtt{a}_2$, and $[\![\forall\alpha{:}\kappa.\tau_1]\!] \equiv \sigma$, and $\mathtt{strip} \sim \mathtt{a}_1$, and $(\Lambda\alpha{:}\kappa.\mathtt{q}_1) \sim \mathtt{a}_2$. Since $\Gamma' \vdash \mathtt{e}' : \tau' \rhd \mathtt{q}'$, we have that $\mathtt{e}' = (\Lambda\alpha{:}\kappa'.\mathtt{e}_1')$, and $\tau'{=}(\forall\alpha{:}\kappa'.\tau_1')$, and $\Gamma',(\alpha{:}\kappa') \vdash \mathtt{e}_1' : \tau_1' : \mathtt{q}_1'$, and $\mathtt{q}' = \mathtt{tabs}\ [\![\forall\alpha{:}\kappa'.\tau_1']\!]\ \mathtt{strip}_{(\mathsf{F},\kappa',[\![\lambda\alpha:\kappa'.\tau 1']\!])}\ (\Lambda\alpha{:}\kappa'.\mathtt{q}_1')$, and $(\Lambda\alpha{:}\kappa.\mathtt{q}_1) \sim (\Lambda\alpha{:}\kappa'.\mathtt{q}_1')$. Therefore, $\kappa{=}\kappa'$ and $\mathtt{q}_1 \sim \mathtt{q}_1'$. Since $\Gamma \sim \Gamma'$ and $\kappa{=}\kappa'$, we have that $\Gamma,(\alpha{:}\kappa) \sim \Gamma',(\alpha{:}\kappa')$. By induction, $\mathtt{e}_1 \sim \mathtt{e}_1'$. Therefore, $(\Lambda\alpha{:}\kappa.\mathtt{e}_1) \sim (\Lambda\alpha{:}\kappa'.\mathtt{e}_1')$ Therefore, $\mathtt{e} \sim \mathtt{e}'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$ is by the fifth rule. Then $\mathtt{e} = (\mathtt{e}_1\ \sigma)$ and $\Gamma \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathtt{q}_1$ and $\tau = (\tau_1[\alpha{:=}\sigma])$, and $\mathtt{q} = \mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{q}_1\ [\![\tau_1[\alpha{:=}\sigma]]\!]\ \mathtt{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],[\![\sigma]\!])}$. From now on we abbreviate $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],[\![\sigma]\!])}$ as $\mathtt{inst}$. Since $\mathtt{q} \sim \mathtt{q}'$, we have that $\mathtt{q}' = \mathtt{tapp}\ \sigma_1\ \mathtt{a}_1\ \sigma_2\ \mathtt{a}_2$, and $[\![\forall\alpha{:}\kappa.\tau_1]\!] \equiv \sigma_1$, and $\mathtt{q}_1 \sim \mathtt{a}_1$, and $[\![\tau_1[\alpha{:=}\sigma]]\!] \equiv \sigma_2$ and $\mathtt{inst} \sim \mathtt{a}_2$.

Since $\Gamma' \vdash \mathtt{e'} : \tau' \rhd \mathtt{q'}$, we have that $\mathtt{e'} = (\mathtt{e}'_1\ \sigma')$, and $\Gamma' \vdash \mathtt{e}'_1 : (\forall\alpha{:}\kappa'.\tau'_1) \rhd \mathtt{q}'_1$, and $\tau' = (\tau'_1[\alpha{:=}\sigma'])$, and $\mathtt{q}'_1 = \mathtt{tapp}\ \llbracket\forall\alpha{:}\kappa'.\tau'_1\rrbracket\ \mathtt{q}'_1\ \llbracket\tau'_1[\alpha{:=}\sigma']\rrbracket\ \mathtt{inst}_{(\llbracket\forall\alpha:\kappa'.\tau1'\rrbracket,\llbracket\sigma'\rrbracket)}$. From now on we abbreviate $\mathtt{inst}_{(\llbracket\forall\alpha:\kappa'.\tau1'\rrbracket,\llbracket\sigma'\rrbracket)}$ as $\mathtt{inst}'$. So we have that $\llbracket\forall\alpha{:}\kappa.\tau_1\rrbracket \equiv \llbracket\forall\alpha{:}\kappa'.\tau'_1\rrbracket$, and $\mathtt{q}_1 \sim \mathtt{q}'_1$, and $\llbracket\tau_1[\alpha{:=}\sigma]\rrbracket \equiv \llbracket\tau'_1[\alpha{:=}\sigma']\rrbracket$, and $\mathtt{inst} \sim \mathtt{inst}'$. By induction $\mathtt{e}_1 \sim \mathtt{e}'_1$. By Theorem 5.3, we have that $(\forall\alpha{:}\kappa.\tau_1) \equiv (\forall\alpha{:}\kappa'.\tau'_1)$ and $(\tau_1[\alpha{:=}\sigma]) \equiv (\tau'_1[\alpha{:=}\sigma'])$. Therefore $\kappa{=}\kappa'$. Since $\mathtt{inst} \sim \mathtt{inst}'$, Lemma A.25 states that $\sigma \equiv \sigma'$. Therefore, $\mathtt{e} \sim \mathtt{e}'$.

Suppose $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$ is by the sixth rule. Then $\Gamma \vdash \mathtt{e} : \tau_1 \rhd \mathtt{q}$. By induction, $\mathtt{e} \sim \mathtt{e}'$. □

Theorem 6.3. *If* $\langle\rangle \vdash \mathtt{e}_1 : \tau$, *and* $\langle\rangle \vdash \mathtt{e}_2 : \tau$, *then* $\mathtt{e}_1 \sim \mathtt{e}_2$ *if and only if* $\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}$.

Proof. $(\mathtt{e}_1 \sim \mathtt{e}_2) \implies (\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2})$: Suppose $\mathtt{e}_1 \sim \mathtt{e}_2$. By Lemma A.24, we have that $\mathtt{q}_1 \sim \mathtt{q}_2$. Therefore, $\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}$.

$(\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}) \implies (\mathtt{e}_1 \sim \mathtt{e}_2)$:

Suppose $\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}$. Then $\mathtt{q}_1 \sim \mathtt{q}_2$. By Lemma A.26, we have that $\mathtt{e}_1 \sim \mathtt{e}_2$.

□

A.4 Section 7 Proofs

Theorem 7.1. $\mathtt{Op\ R}\ \overline{\tau} \equiv \mathtt{R}\ (\llbracket\tau\rrbracket\mathtt{[F := R]})$.

Proof.

$$\begin{aligned}
&\mathtt{Op\ R}\ \overline{\tau}\\
=\ &(\lambda\mathtt{F}{:}{*} \to {*}.\ \lambda\alpha{:}\mathtt{U}.\ \alpha\ \mathtt{F})\ \mathtt{R}\ \overline{\tau}\\
\equiv\ &\mathtt{R}\ (\overline{\tau}\ \mathtt{R})\\
=\ &\mathtt{R}\ ((\lambda\mathtt{F}{:}{*} \to {*}.\ \llbracket\tau\rrbracket)\ \mathtt{R})\\
\equiv\ &\mathtt{R}\ (\llbracket\tau\rrbracket\mathtt{[F := R]}).
\end{aligned}$$

□

Theorem 7.2. *If* $\mathtt{f = foldExp\ R\ abs'\ app'\ tabs'\ tapp'}$, *and* $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, *then* $\mathtt{f}\ \overline{\tau}\ \overline{\mathtt{e}}.\ \longrightarrow^{*} (\mathtt{q[F{:=}R,\ abs{:=}abs',\ app{:=}app',\ tabs{:=}tabs',\ tapp{:=}tapp']})$.

Proof. Suppose $\mathtt{f = foldExp\ R\ abs'\ app'\ tabs'\ tapp'}$, and $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$. Then,

$$\begin{aligned}
&\mathtt{f}\ \overline{\tau}\ \overline{\mathtt{e}}\\
=\ &(\Lambda\mathtt{F}{:}{*} \to {*}\\
&\quad\lambda\mathtt{abs{:}Abs\ F}.\ \lambda\mathtt{app{:}App\ F}.\\
&\quad\lambda\mathtt{tabs{:}TAbs\ F}.\ \lambda\mathtt{tapp{:}TApp\ F}.\\
&\quad\Lambda\alpha{:}\mathtt{U}.\ \lambda\mathtt{e{:}Exp}\ \alpha.\\
&\quad\mathtt{e\ F\ abs\ app\ tabs\ tapp})\\
&\mathtt{R\ abs'\ app'\ tabs'\ tapp'}\ \overline{\tau}\ \overline{\mathtt{e}}\\
\longrightarrow^{7}\ &\overline{\mathtt{e}}\ \mathtt{R\ abs'\ app'\ tabs'\ tapp'}\\
=\ &(\Lambda\mathtt{F}{:}{*} \to {*}.\\
&\quad\lambda\mathtt{abs{:}Abs\ F}.\ \lambda\mathtt{app{:}App\ F}.\\
&\quad\lambda\mathtt{tabs{:}TAbs\ F}.\ \lambda\mathtt{tapp{:}TApp\ F}.\\
&\quad\mathtt{q})\ \mathtt{R\ abs'\ app'\ tabs'\ tapp'}\\
\longrightarrow^{5}\ &(\mathtt{q[F{:=}R,\ abs{:=}abs',\ app{:=}app',}\\
&\quad\mathtt{tabs{:=}tabs',\ tapp{:=}tapp']})
\end{aligned}$$

□

Lemma A.27. $(\llbracket\tau\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \tau$.

Proof. By induction on the structure of τ.

Suppose τ is a type variable α. Then $\llbracket\tau\rrbracket = \alpha$, so $\llbracket\tau\rrbracket\mathtt{[F := Id]} = (\alpha\mathtt{[F := Id]}) = \alpha$, as required.

Suppose τ is an arrow type $\tau_1 \to \tau_2$. Then $\llbracket\tau\rrbracket = \mathtt{F}\ \llbracket\tau_1\rrbracket \to \mathtt{F}\ \llbracket\tau_2\rrbracket$, so $(\llbracket\tau\rrbracket\mathtt{[F := Id]}) = (\mathtt{F}\ \llbracket\tau_1\rrbracket \to \mathtt{F}\ \llbracket\tau_2\rrbracket)\mathtt{[F := Id]} = \mathtt{Id}\ (\llbracket\tau_1\rrbracket\mathtt{[F := Id]}) \to \mathtt{Id}\ (\llbracket\tau_2\rrbracket\mathtt{[F := Id]})$. By induction, $(\llbracket\tau_1\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \tau_1$, and $(\llbracket\tau_2\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \tau_2$, so $\llbracket\tau\rrbracket\mathtt{[F := Id]} \longrightarrow^{*} \mathtt{Id}\ \tau_1 \to \mathtt{Id}\ \tau_2 = (\lambda\alpha{:}{*}.\ \alpha)\ \tau_1 \to (\lambda\alpha{:}{*}.\ \alpha)\ \tau_2 \longrightarrow^{2} \tau_1 \to \tau_2 = \tau$, as required.

Suppose τ is a quantified type $\forall\alpha{:}\kappa.\tau_1$. Then $\llbracket\tau\rrbracket = \forall\alpha{:}\kappa.\ \mathtt{F}\ \llbracket\tau_1\rrbracket$, so $\llbracket\tau\rrbracket\mathtt{[F := Id]} = ((\forall\alpha{:}\kappa.\ \mathtt{F}\ \llbracket\ \tau_1\rrbracket)\mathtt{[F := Id]}) = (\forall\alpha{:}\kappa.\ \mathtt{Id}\ (\llbracket\tau_1\rrbracket\mathtt{[F := Id]}))$. By induction, $(\llbracket\tau_1\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \tau_1$, so $(\llbracket\tau\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \forall\alpha{:}\kappa.\ \mathtt{Id}\ \tau_1 = \forall\alpha{:}\kappa.\ (\lambda\alpha{:}{*}.\alpha)\ \tau_1 \longrightarrow \forall\alpha{:}\kappa.\tau_1 = \tau$, as required.

Suppose τ is a λ-abstraction $\lambda\alpha{:}\kappa.\tau_1$. Then $\llbracket\tau\rrbracket = \lambda\alpha{:}\kappa.\llbracket\tau_1\rrbracket$, so $\llbracket\tau\rrbracket\mathtt{[F := Id]} = (\lambda\alpha{:}\kappa.(\llbracket\tau_1\rrbracket\mathtt{[F := Id]}))$. By induction, $(\llbracket\tau_1\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \tau_1$, so $(\llbracket\tau\rrbracket\mathtt{[F := Id]}) = \lambda\alpha{:}\kappa.\tau_1 = \tau$, as required.

Suppose τ is a type application $\tau_1\ \tau_2$. Then $\llbracket\tau\rrbracket = \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket$, so $\llbracket\tau\rrbracket\mathtt{[F := Id]} = (\llbracket\tau_1\rrbracket\mathtt{[F := Id]})\ (\llbracket\tau_2\rrbracket\mathtt{[F := Id]})$. By induction, $(\llbracket\tau_1\rrbracket\mathtt{[F := Id]}) \longrightarrow^{*} \tau_1$, and $\llbracket\tau_2\rrbracket\mathtt{[F := Id]} \longrightarrow^{*} \tau_2$, so $\llbracket\tau\rrbracket\mathtt{[F := Id]} \longrightarrow^{*} \tau_1\ \tau_2 = \tau$, as required. □

Theorem 7.3. *If* $\Gamma \vdash \tau : *$, *then* $\mathtt{Op\ Id}\ \overline{\tau} \equiv \tau$.

Proof. By Theorem 7.1, $\mathtt{Op\ Id}\ \overline{\tau} \equiv \mathtt{Id}\ (\llbracket\tau\rrbracket\mathtt{[F := Id]}) \equiv \llbracket\tau\rrbracket\mathtt{[F := Id]}$. By Lemma A.27, $(\llbracket\tau\rrbracket\mathtt{[F := Id]}) \equiv \tau$, as required. □

We introduce new notation for an abstract substitutions: the application of substitution θ to a term $\mathtt{e}$ is written $\theta(\mathtt{e})$. We write $\theta\mathtt{[x{:=}e]}$ to denote an extention of θ. In particular, $(\theta\mathtt{[x{:=}e_1]})(\mathtt{e}_2) = \theta(\mathtt{e}_2\mathtt{[x{:=}e_1]})$.

Lemma A.28. *If* $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, *then* $\mathtt{q[F := Id,\ abs := unAbs,\ app := unApp,\ tabs := unTAbs,\ tapp := unTApp]} \longrightarrow^{*}\mathtt{e}$.

Proof. Suppose that $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$. Let $\theta = \mathtt{[F := Id,\ abs := unAbs,\ app := unApp,\ tabs := unTAbs,\ tapp := unTApp]}$. We want to show that $\theta(\mathtt{q}) \longrightarrow^{*} \mathtt{e}$. We proceed by induction on the structure of $\mathtt{e}$.

Suppose that $\mathtt{e}$ is a variable $\mathtt{x}$. Then $\mathtt{q} = \mathtt{x}$, so $\theta(\mathtt{q}) = \mathtt{x}$ as required.

Suppose that $\mathtt{e}$ is a λ-abstraction $\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1$, and that $\tau{=}\tau_1 \to \tau_2$. Then $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{abs}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ (\lambda\mathtt{x{:}F}\ \llbracket\tau_1\rrbracket.\ \mathtt{q}_1)$. By induction, we have that $\theta(\mathtt{q}_1) \longrightarrow^{*} \mathtt{e}_1$. By Lemma A.27, $\theta(\llbracket\tau_1\rrbracket) \longrightarrow^{*}\tau_1$ and $\theta(\llbracket\tau_2\rrbracket) \longrightarrow^{*}\tau_2$. Therefore,

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{abs}\ \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket\ (\lambda\mathtt{x{:}F}\ \llbracket\tau_1\rrbracket.\ \mathtt{q}_1))\\
&= \mathtt{unAbs}\ \theta(\llbracket\tau_1\rrbracket)\ \theta(\llbracket\tau_2\rrbracket)\ (\lambda\mathtt{x{:}Id}\ \theta(\llbracket\tau_1\rrbracket).\ \theta(\mathtt{q}_1))\\
\longrightarrow^{*}\ &\mathtt{unAbs}\ \tau_1\ \tau_2\ (\lambda\mathtt{x}{:}\tau_1.\ \mathtt{e}_1)\\
&= (\Lambda\alpha{:}{*}.\ \Lambda\beta{:}{*}.\ \lambda\mathtt{f}{:}\ \alpha \to \beta.\ \mathtt{f})\ \tau_1\ \tau_2\ (\lambda\mathtt{x}{:}\tau_1.\ \mathtt{e}_1)\\
\longrightarrow^{3}\ &(\lambda\mathtt{x}{:}\tau_1.\ \mathtt{e}_1)\\
&= \mathtt{e}.
\end{aligned}$$

Suppose that $\mathtt{e}$ is an application $\mathtt{e}_1\ \mathtt{e}_2$. Then $\Gamma \vdash \mathtt{e}_1 : \tau_1 \to \tau \rhd \mathtt{q}_1$, and $\Gamma \vdash \mathtt{e}_2 : \tau_1 \rhd \mathtt{q}_2$, and cq = app $\llbracket\tau_1\rrbracket\ \llbracket\tau\rrbracket\ \mathtt{q}_1\ \mathtt{q}_2$. By Lemma A.27, $\theta(\llbracket\tau_1\rrbracket) \longrightarrow^{*}\tau_1$ and $\theta(\llbracket\tau\rrbracket) \longrightarrow^{*}\tau$. By induction, $\theta(\mathtt{q}_1) \longrightarrow^{*}\mathtt{e}_1$ and $\theta(\mathtt{q}_2) \longrightarrow^{*}\mathtt{e}_2$. Therefore,

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{app}\ \llbracket\tau_1\rrbracket\ \llbracket\tau\rrbracket\ \mathtt{q}_1\ \mathtt{q}_2)\\
&= \mathtt{unApp}\ \theta(\llbracket\tau_1\rrbracket)\ \theta(\llbracket\tau\rrbracket)\ \theta(\mathtt{q}_1)\ \theta(\mathtt{q}_2)\\
\longrightarrow^{*}\ &\mathtt{unApp}\ \tau_1\ \tau\ \mathtt{e}_1\ \mathtt{e}_2\\
&= (\Lambda\alpha{:}{*}.\ \Lambda\beta{:}{*}.\ \lambda\mathtt{f}{:}\ \alpha \to \beta.\ \mathtt{f})\ \tau_1\ \tau_2\ \mathtt{e}_1\ \mathtt{e}_2\\
\longrightarrow^{3}\ &\mathtt{e}_1\ \mathtt{e}_2\\
&= \mathtt{e}.
\end{aligned}$$

Suppose that $\mathtt{e}$ is a Λ-abstraction $\Lambda\alpha{:}\kappa.\mathtt{e}_1$. Then $\tau = \forall\alpha{:}\kappa.\tau_1$, and $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{tabs}\ \tau_2\ \mathtt{strip}\ (\Lambda\alpha{:}\kappa.\mathtt{q}_1)$, for some some term $\mathtt{strip}$. By induction, $\theta(\mathtt{q}_1) \longrightarrow^{*}\mathtt{e}_1$. Therefore,

$\theta(\mathtt{q})$
$= \theta(\mathtt{tabs}\ \tau_2\ \mathtt{strip}\ (\Lambda\alpha{:}\kappa.\ \mathtt{q}_1))$
$= \mathtt{unTAbs}\ \theta(\tau_2)\ \theta(\mathtt{strip})\ (\Lambda\alpha{:}\kappa.\ \theta(\mathtt{q}_1))$
$\longrightarrow^*\ \mathtt{unTAbs}\ \theta(\tau_2)\ \theta(\mathtt{strip})\ (\Lambda\alpha{:}\kappa.\ \mathtt{e}_1)$
$= (\Lambda\alpha{:}{*}.\ \lambda\mathtt{s}{:}\mathtt{Strip}\ \mathtt{F}\ \alpha.\ \lambda\mathtt{f}{:}\alpha.\ \mathtt{f})$
$\quad \tau_2\ \mathtt{strip}\ (\Lambda\alpha{:}\kappa.\ \mathtt{e}_1)$
$\longrightarrow^3\ (\Lambda\alpha{:}\kappa.\ \mathtt{e}_1)$
$= \mathtt{e}.$

Suppose that $\mathtt{e}$ is a type application $\mathtt{e}_1\ \sigma$. Then $\Gamma \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau)$, and $\Gamma \vdash \sigma : \kappa$, and $\tau = \tau_1[\alpha{:=}\sigma]$, and $\mathtt{q} = \mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{q}_1\ [\![\tau]\!]\ (\lambda\mathtt{x}{:}[\![\forall\alpha{:}\kappa.\tau_1]\!].\ \mathtt{x}\ [\![\sigma]\!])$ By Lemma A.27, $\theta([\![\forall\alpha{:}\kappa.\tau_1]\!]) \longrightarrow^* [\![\forall\alpha{:}\kappa.\tau_1]\!]$, and $\theta([\![\sigma]\!]) \longrightarrow^* \sigma$, and $\theta([\![\tau_1[\alpha{:=}\sigma]]\!]) \longrightarrow^* (\tau_1[\alpha{:=}\sigma])$. By induction, $\theta(\mathtt{q}_1) \longrightarrow^* \mathtt{e}_1$. Therefore,

$\theta(\mathtt{q})$
$= \theta(\mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{q}_1\ [\![\tau_1[\alpha{:=}\sigma]]\!]$
$\quad (\lambda\mathtt{x}{:}[\![\forall\alpha{:}\kappa.\tau]\!].\ \mathtt{x}\ [\![\sigma]\!]))$
$= \mathtt{unTApp}\ \theta([\![\forall\alpha{:}\kappa.\tau_1]\!])\ \theta(\mathtt{q}_1)\ \theta([\![\tau_1[\alpha{:=}\sigma]]\!])$
$\quad (\lambda\mathtt{x}{:}\theta([\![\forall\alpha{:}\kappa.\tau]\!]).\ \mathtt{x}\ \theta([\![\sigma]\!])))$
$\longrightarrow^*\ \mathtt{unTApp}\ (\forall\alpha{:}\kappa.\tau_1)\ \mathtt{e}_1\ (\tau_1[\alpha{:=}\sigma])$
$\quad (\lambda\mathtt{x}{:}(\forall\alpha{:}\kappa.\tau).\ \mathtt{x}\ \sigma)$
$= (\Lambda\alpha{:}{*}.\ \lambda\mathtt{f}{:}\alpha.\ \Lambda\beta{:}{*}.\ \lambda\mathtt{g}{:}\alpha \to \beta.\ \mathtt{g}\ \mathtt{f})$
$\quad (\forall\alpha{:}\kappa.\tau_1)\ \mathtt{e}_1\ (\tau_1[\alpha{:=}\sigma])\ (\lambda\mathtt{x}{:}(\forall\alpha{:}\kappa.\tau).\ \mathtt{x}\ \sigma)$
$\longrightarrow^4\ (\lambda\mathtt{x}{:}(\forall\alpha{:}\kappa.\tau).\ \mathtt{x}\ \sigma)\ \mathtt{e}_1$
$\longrightarrow^1\ \mathtt{e}_1\ \sigma$
$= \mathtt{e}$

□

Theorem 7.4. *If* $\langle\rangle \vdash \mathtt{e} : \tau$, *then* $\mathtt{unquote}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{e}$.

Proof. Suppose $\langle\rangle \vdash \mathtt{e} : \tau$. Let $\mathtt{q}$ be such that $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$. By Theorem 7.2, we have that $\mathtt{unquote}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{q}$`[F := Id, abs := unAbs, app := unApp, tabs := unTAbs, tapp := unTApp]`.

By Lemma A.28, `q[F := Id, abs := unAbs, app := unApp, tabs := unTAbs, tapp := unTApp]` $\longrightarrow^* \mathtt{e}$. Therefore $\mathtt{unquote}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{e}$. □

Theorem 7.5. *If* $\Gamma \vdash \tau : *$, *then* $\mathtt{Op}\ \mathtt{KBool}\ \overline{\tau} \equiv \mathtt{Bool}$.

Proof. By Theorem 7.1, we have that $\mathtt{Op}\ \mathtt{KBool}\ \overline{\tau} \equiv \mathtt{KBool}\ ([\![\tau]\!][\mathtt{F} := \mathtt{KBool}]) = (\lambda\alpha{:}{*}.\ \mathtt{Bool})\ ([\![\tau]\!][\mathtt{F} := \mathtt{KBool}]) \equiv \mathtt{Bool}$. □

Theorem 7.6. *Suppose* $\langle\rangle \vdash \mathtt{e} : \tau$. *If* $\mathtt{e}$ *is an abstraction then* $\mathtt{isAbs}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{true}$. *Otherwise* $\mathtt{e}$ *is an application and* $\mathtt{isAbs}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{false}$.

Proof. Suppose $\langle\rangle \vdash \mathtt{e} : \tau$. Let $\mathtt{q}$ be such that $\langle\rangle \vdash \mathtt{e} : \tau \rhd \mathtt{q}$. By Theorem 7.2, we have that $\mathtt{isAbs}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{q}$`[F := KBool, abs := isAbsAbs, app := isAbsApp, tabs := isAbsTAbs, tapp := isAbsTApp]`.

Let θ = `[F := KBool, abs := isAbsAbs, app := isAbsApp, tabs := isAbsTAbs, tapp := isAbsTApp]`.

We now want to show that $\theta(\mathtt{q}) \longrightarrow^* \mathtt{true}$ if $\mathtt{e}$ is an abstraction, and that $\theta(\mathtt{q}) \longrightarrow^* \mathtt{false}$ if $\mathtt{e}$ is an application. We proceed by case analysis on the structure of $\mathtt{e}$.

Suppose $\mathtt{e}$ is a variable $\mathtt{x}$. Then $(\mathtt{x}{:}\tau) \in \langle\rangle$, which is a contradiction. Therefore our assumption that $\mathtt{e}$ is a variable is false.

Suppose $\mathtt{e}$ is a λ-abstraction $(\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1)$. Then $\tau = \tau_1 \to \tau_2$, and $\langle\rangle,\mathtt{x}{:}\tau_1 \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$, and

$\theta(\mathtt{q})$
$= \theta(\mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1))$
$= \mathtt{isAbsAbs}\ \theta([\![\tau_1]\!])\ \theta([\![\tau_2]\!])\ \theta(\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$
$= (\Lambda\alpha{:}{*}.\ \Lambda\beta{:}{*}.\ \lambda\mathtt{f}{:}\mathtt{Bool} \to \mathtt{Bool}.\ \mathtt{true})$
$\quad \theta([\![\tau_1]\!])\ \theta([\![\tau_2]\!])\ \theta(\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$
$\longrightarrow^3\ \mathtt{true}$

Suppose $\mathtt{e}$ is an application $\mathtt{e}_1\ \mathtt{e}_2$. Then $\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau \rhd \mathtt{q}_1$, and $\Gamma \vdash \mathtt{e}_2 : \tau_2 \rhd \mathtt{q}_2$, and

$\theta(\mathtt{q})$
$= \theta(\mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2)$
$= \mathtt{isAbsApp}\ \theta([\![\tau_2]\!])\ \theta([\![\tau]\!])\ \theta(\mathtt{q}_1)\ \theta(\mathtt{q}_2)$
$= (\Lambda\alpha{:}{*}.\ \Lambda\beta{:}{*}.\ \lambda\mathtt{f}{:}\mathtt{Bool}.\ \lambda\mathtt{x}{:}\mathtt{Bool}.\ \mathtt{false})$
$\quad \theta([\![\tau_2]\!])\ \theta([\![\tau]\!])\ \theta(\mathtt{q}_1)\ \theta(\mathtt{q}_2)$
$\longrightarrow^4\ \mathtt{false}$

If $\mathtt{e}$ is a type abstraction $(\Lambda\alpha{:}\kappa.\mathtt{e}_1)$, then $\tau = (\forall\alpha{:}\kappa.\tau_1)$, and $\langle\rangle,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and

$\theta(\mathtt{q})$
$= \theta(\mathtt{tabs}\ [\![\tau]\!]\ \mathtt{strip}\ (\Lambda\alpha{:}\kappa.\mathtt{q}_1))$
$= \mathtt{isAbsTAbs}\ \theta([\![\tau]\!])\ \theta(\mathtt{strip})\ \theta(\Lambda\alpha{:}\kappa.\mathtt{q}_1)$
$= (\Lambda\alpha{:}{*}.\ \lambda\mathtt{strip}{:}\mathtt{Strip}\ \mathtt{KBool}\ \alpha.\ \lambda\mathtt{f}{:}\alpha.\ \mathtt{true})$
$\quad \theta([\![\tau]\!])\ \theta(\mathtt{strip})\ \theta(\Lambda\alpha{:}\kappa.\mathtt{q}_1)$
$\longrightarrow^3\ \mathtt{true}$

Suppose $\mathtt{e}$ is a type application $\mathtt{e}_1\ \sigma$. Then $\langle\rangle \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathtt{q}_1$, and $\tau = (\tau_1[\alpha{:=}\sigma])$. Therefore,

$\theta(\mathtt{q})$
$= \theta(\mathtt{tapp}\ [\![\tau_1]\!]\ \mathtt{q}_1\ [\![\tau]\!]\ \mathtt{inst})$
$= \mathtt{isAbsTApp}\ \theta([\![\tau_1]\!])\ \theta(\mathtt{q}_1)\ \theta([\![\tau]\!])\ \theta(\mathtt{inst})$
$= (\Lambda\alpha{:}{*}.\ \lambda\mathtt{f}{:}\mathtt{Bool}.\ \Lambda\beta{:}{*}.\ \lambda\mathtt{inst}{:}\alpha \to \mathtt{Bool}.$
$\quad \mathtt{false})$
$\quad \theta([\![\tau_1]\!])\ \theta(\mathtt{q}_1)\ \theta([\![\tau]\!])\ \theta(\mathtt{inst})$
$\longrightarrow^4\ \mathtt{false}$

□

Theorem 7.7. *If* $\Gamma \vdash \tau : *$, *then* $\mathtt{Op}\ \mathtt{KNat}\ \overline{\tau} \equiv \mathtt{Nat}$.

Proof. Similar to the proof of Theorem 7.5. □

Lemma A.29. $\mathtt{church}_n \longrightarrow^* (\Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}^n\ \mathtt{z})$.

Proof. By induction on n.

If $n{=}0$, then $\mathtt{church}_n = \mathtt{zero} = (\Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{z}) = (\Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}^0\ \mathtt{z})$.

Suppose $n{>}0$. By induction, $\mathtt{church}_{n-1} \longrightarrow^* (\Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}^{n-1}\ \mathtt{z})$. We have that $\mathtt{church}_n = \mathtt{succ}\ \mathtt{church}_{n-1} \longrightarrow^* \mathtt{succ}\ (\Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}^{n-1}\ \mathtt{z}) \longrightarrow \Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}\ ((\Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}^{n-1}\ \mathtt{z})\ \alpha\ \mathtt{z}\ \mathtt{s}) \longrightarrow^3 \Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}\ (\mathtt{s}^{n-1}\ \mathtt{z}) = \Lambda\alpha{:}{*}.\ \lambda\mathtt{z}{:}\alpha.\ \lambda\mathtt{s}{:}\alpha \to \alpha.\ \mathtt{s}^n\ \mathtt{z}$ □

Lemma A.30. $\mathtt{plus}\ \mathtt{church}_m\ \mathtt{church}_n \longrightarrow^* \mathtt{church}_{m+n}$.

Proof.

$\mathtt{plus}\ \mathtt{church}_m\ \mathtt{church}_n$
$= \mathtt{church}_m\ \mathtt{Nat}\ \mathtt{church}_n\ \mathtt{succ}$
$\longrightarrow^*\ \mathtt{succ}^m\ \mathtt{church}_n$
$= \mathtt{church}_{m+n}$

□

Lemma A.31. *If* $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, *and* Γ *binds term variables* $\mathtt{x}_1, \ldots, \mathtt{x}_n$, *and* θ=`[F := KNat, abs := sizeAbs, app := sizeApp, tabs := sizeTAbs, tapp := sizeTApp,` $\mathtt{x}_1 := \mathtt{church}_1$, $\mathtt{x}_2 := \mathtt{church}_1$, $\ldots$ $\mathtt{x}_n := \mathtt{church}_1$`]`, *then* $\theta(\mathtt{q}) \longrightarrow^* \mathtt{church}_{|\mathtt{e}|}$.

Proof. Suppose $\Gamma \vdash \mathtt{e} : \tau : \mathtt{q}$, and θ is of the required form for Γ.

We proceed by induction on the structure of the derivation $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$.

Suppose $\mathtt{e}$ is a variable $\mathtt{x}$. Then $|\mathtt{e}| = 1$ and $(x{:}\tau) \in \Gamma$, so θ maps $\mathtt{x}$ to $\mathtt{church}_1$. We have that $\theta(\mathtt{q}) = \theta(\mathtt{x}) = \mathtt{church}_1$ as required.

Suppose $\mathtt{e}$ is an abstraction $(\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1)$. Then $\tau = \tau_1 \to \tau_2$, and $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$, and Let $\theta' = \theta[\mathrm{x} := \mathrm{church}_1]$. Then θ' is of the required form for $\Gamma,(\mathtt{x}{:}\tau_1)$. By induction, $\theta'(\mathtt{q}_1) \longrightarrow^* \mathtt{church}_{|\mathtt{e1}|}$. Therefore,

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\mathtt{q}_1))\\
&= \mathtt{sizeAbs}\ \theta([\![\tau_1]\!])\ \theta([\![\tau_2]\!])\ (\lambda\mathtt{x}{:}\mathtt{KNat}\ \theta([\![\tau_1]\!]).\theta(\mathtt{q}_1))\\
\longrightarrow^* &\ \mathtt{succ}\ ((\lambda\mathtt{x}{:}\mathtt{Nat}.\theta(\mathtt{q}_1))\ (\mathtt{succ\ zero}))\\
\longrightarrow^* &\ \mathtt{succ}\ \theta(\mathtt{q}_1[\mathtt{x} := \mathtt{succ\ zero}])\\
&= \mathtt{succ}\ \theta(\mathtt{q}_1[\mathtt{x} := \mathtt{succ\ zero}])\\
&= \mathtt{succ}\ (\theta[\mathtt{x} := \mathtt{succ\ zero}])(\mathtt{q}_1)\\
&= \mathtt{succ}\ (\theta[\mathtt{x} := \mathtt{church}_1])(\mathtt{q}_1)\\
&= \mathtt{succ}\ \theta'(\mathtt{q}_1)\\
\longrightarrow^* &\ \mathtt{succ\ church}_{|\mathtt{e1}|}\\
&= \mathtt{church}_{1+|\mathtt{e1}|}\\
&= \mathtt{church}_{|\mathtt{e}|}.
\end{aligned}$$

Suppose $\mathtt{e}$ is an application $\mathtt{e}_1\ \mathtt{e}_2$. Then $\Gamma \vdash \mathtt{e}_1 : \tau_2 \to \tau : \mathtt{q}_1$ and $\Gamma \vdash \mathtt{e}_2 : \tau_2 : \mathtt{q}_2$. By induction, $\theta(\mathtt{q}_1) \longrightarrow^* \mathtt{church}_{|\mathtt{e1}|}$ and $\theta(\mathtt{q}_2) \longrightarrow^* \mathtt{church}_{|\mathtt{e2}|}$. Therefore,

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{app}\ [\![\tau_2]\!]\ [\![\tau]\!]\ \mathtt{q}_1\ \mathtt{q}_2)\\
&= \mathtt{sizeApp}\ \theta([\![\tau_2]\!])\ \theta([\![\tau]\!])\ \theta(\mathtt{q}_1)\ \theta(\mathtt{q}_2)\\
\longrightarrow^* &\ \mathtt{succ}\ (\mathtt{plus}\ \theta(\mathtt{q}_1)\ \theta(\mathtt{q}_2))\\
\longrightarrow^* &\ \mathtt{succ}\ (\mathtt{plus\ church}_{|\mathtt{e1}|}\ \mathtt{church}_{|\mathtt{e2}|})\\
\longrightarrow^* &\ \mathtt{succ\ church}_{|\mathtt{e1}|+|\mathtt{e2}|} \quad \text{(By Lemma A.30)}\\
&= \mathtt{church}_{1+|\mathtt{e1}|+|\mathtt{e2}|}\\
&= \mathtt{church}_{|\mathtt{e1\ e2}|}.
\end{aligned}$$

Suppose $\mathtt{e}$ is a type abstraction $\Lambda\alpha{:}\kappa.\mathtt{e}_1$. Then $\tau = (\forall\alpha{:}\kappa.\tau_1)$, and $\Gamma,(\alpha{:}\kappa) \vdash \mathtt{e}_1 : \tau_1 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{tabs}\ [\![\tau]\!]\ \mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:kappa.\tau1]\!])}\ (\Lambda\alpha{:}\kappa.\mathtt{q}_1)$. From now on we use $\mathtt{strip}$ to refer to $\mathtt{strip}_{(\mathtt{F},\kappa,[\![\lambda\alpha:kappa.\tau1]\!])}$. We have that θ is of the required form for $\Gamma,(\alpha{:}\kappa)$, so by induction, $\theta(\mathtt{q}_1) \longrightarrow^* \mathtt{church}_{|\mathtt{e1}|}$. By Lemma A.11, we have that $(\theta(\mathtt{strip})\ \mathtt{Nat}\ (\Lambda\alpha{:}*.\ \lambda\mathtt{x}{:}\mathtt{Nat}.\ \mathtt{x})\ (\Lambda\alpha{:}\kappa.\mathtt{church}_{|\mathtt{e1}|}))$ $\longrightarrow^* \mathtt{church}_{|\mathtt{e1}|}[\alpha{:=}\sigma] = \mathtt{church}_{|\mathtt{e1}|}$, where $\kappa \blacktriangleright \sigma$. Now, we have that

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{tabs}\ [\![\tau]\!]\ \mathtt{strip}\ (\Lambda\alpha{:}\kappa.\mathtt{q}_1))\\
&= \mathtt{sizeTAbs}\ \theta([\![\tau]\!])\ \theta(\mathtt{strip})\ (\Lambda\alpha{:}\kappa.\theta(\mathtt{q}_1))\\
\longrightarrow^* &\ \mathtt{succ}\ (\theta(\mathtt{strip})\ \mathtt{Nat}\ (\Lambda\alpha{:}*.\ \lambda\mathtt{x}{:}\mathtt{Nat}.\ \mathtt{x})\\
&\qquad (\Lambda\alpha{:}\kappa.\theta(\mathtt{q}_1)))\\
\longrightarrow^* &\ \mathtt{succ}\ (\theta(\mathtt{strip})\ \mathtt{Nat}\ (\Lambda\alpha{:}*.\ \lambda\mathtt{x}{:}\mathtt{Nat}.\ \mathtt{x})\\
&\qquad (\Lambda\alpha{:}\kappa.\mathtt{church}_{|\mathtt{e1}|}))\\
\longrightarrow^* &\ \mathtt{succ}\ (\mathtt{church}_{|\mathtt{e1}|})\\
&= \mathtt{church}_{1+|e1|}\\
&= \mathtt{church}_{|e|}.
\end{aligned}$$

Suppose $\mathtt{e}$ is a type application $\mathtt{e}_1\ \sigma$. Then $\Gamma \vdash \mathtt{e}_1 : (\forall\alpha{:}\kappa.\tau_1) \rhd \mathtt{q}_1$ and $\mathtt{q} = \mathtt{sizeTApp}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{q}_1\ [\![\tau_1[\alpha{:=}\sigma]]\!]$ $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau]\!],[\![\sigma]\!])}$. From now on, we use $\mathtt{inst}$ to refer to $\mathtt{inst}_{([\![\forall\alpha:\kappa.\tau]\!],[\![\sigma]\!])}$. By induction, $\theta(\mathtt{q}_1) \longrightarrow^* \mathtt{church}_{|e1|}$. Therefore,

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{tapp}\ [\![\forall\alpha{:}\kappa.\tau_1]\!]\ \mathtt{q}_1\ [\![\tau_1[\alpha{:=}\sigma]]\!]\ \mathtt{inst})\\
&= \mathtt{sizeTApp}\ \theta([\![\forall\alpha{:}\kappa.\tau_1]\!])\ \theta(\mathtt{q}_1)\\
&\qquad \theta([\![\tau_1[\alpha{:=}\sigma]]\!])\ \theta(\mathtt{inst})\\
\longrightarrow^* &\ \mathtt{succ}\ \theta(\mathtt{q}_1)\\
\longrightarrow^* &\ \mathtt{succ\ church}_{|e1|}\\
&= \mathtt{church}_{1+|e1|}\\
&= \mathtt{church}_{|e|}.
\end{aligned}$$

□

Theorem 7.8.
If $\langle\rangle \vdash \mathtt{e} : \tau$ *and* $|\mathtt{e}|{=}n$, *then* $\mathtt{size}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{church}_n$

Proof. Now, suppose $\langle\rangle \vdash \mathtt{e} : \tau$. By Theorem 7.2, we have that $\mathtt{size}\ \overline{\tau}\ \overline{\mathtt{e}} \longrightarrow^* \mathtt{q}[\mathtt{F := KNat,\ abs := sizeAbs,\ app := sizeApp,\ tabs := sizeTAbs,\ tapp := sizeTApp}]$. Let $\theta = [\mathtt{F := KNat,\ abs := sizeAbs,\ app := sizeApp,\ tabs := sizeTAbs,\ tapp := sizeTApp}]$. Since $\mathtt{e}$ is closed, the θ maps each free variable in $\mathtt{e}$ to $\mathtt{church}_1$. Therefore, by Lemma A.31, $\theta(\mathtt{q}) \longrightarrow^* \mathtt{church}_{|\mathtt{e}|}$.

□

Theorem 7.9. *If* $\Gamma \vdash \tau : *$, *then* $\mathtt{Op\ KBools}\ \overline{\tau} \equiv \mathtt{Bools}$.

Proof. Similar to the proof of Theorem 7.5. □

Lemma A.32. *1. If* $\lambda\mathtt{x}{:}\tau.\mathtt{e}$ *is not normal, then* $\mathtt{e}$ *is not normal.*
2. If $\Lambda\alpha{:}\kappa.\mathtt{e}$ *is not normal, then* $\mathtt{e}$ *is not normal.*
3. If $\mathtt{e}_1\ \mathtt{e}_2$ *is normal, then* $\mathtt{e}_1\ \mathtt{e}_2$ *is neutral.*
4. If $\mathtt{e}\ \tau$ *is normal, then* $\mathtt{e}\ \tau$ *is neutral.*
5. If $\mathtt{e}\ \tau$ *is not normal, then* $\mathtt{e}$ *is not normal and neutral.*

Proof. (1) Suppose $(\lambda\mathtt{x}{:}\tau.\mathtt{e})$ is not normal, and suppose further that $\mathtt{e}$ is normal. Then $(\lambda\mathtt{x}{:}\tau.\mathtt{e})$ is normal. Contradiction. Therefore, our assumption $\mathtt{e}$ is normal is false.

(2) Similar to (1).

(3) Suppose $\mathtt{e}_1\ \mathtt{e}_2$ is normal. Then $\mathtt{e}_1$ is normal and neutral, and $\mathtt{e}_2$ is normal. Therefore, $\mathtt{e}_1\ \mathtt{e}_2$ is normal and neutral.

(4) Similar to (3).

(5) Suppose $\mathtt{e}\ \tau$ is not normal. Suppose further that $\mathtt{e}$ is normal. Then $\mathtt{e}\ \tau$ is normal. Contradiction. Therefore, our assumption $\mathtt{e}$ is normal is false. □

Lemma A.33. *If* $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, *and* Γ *binds terms variables* $\mathtt{x}_1 \ldots \mathtt{x}_n$, *and* $\theta{=}[\mathtt{F := KNat,\ abs := nfAbs,\ app := nfApp,\ tabs := nfTAbs,\ tapp := nfTApp,\ x_1 := bools\ true\ true,\ x_2 := bools\ true\ true,}\ \ldots\ \mathtt{x}_n \mathtt{\ := bools\ true\ true}]$, *then one of the following is true:*

1. $\mathtt{e}$ *is normal and neutral and* $\theta(\mathtt{q}) \longrightarrow^* \mathtt{bools\ true\ true}$.
2. $\mathtt{e}$ *is normal but not neutral and* $\theta(\mathtt{q}) \longrightarrow^* \mathtt{bools\ true\ false}$.
3. $\mathtt{e}$ *is not normal and* $\theta(\mathtt{q}) \longrightarrow^* \mathtt{bools\ false\ false}$.

Proof. Suppose $\Gamma \vdash \mathtt{e} : \tau : \mathtt{q}$, and θ is of the required form for Γ.

We proceed by induction on the structure of the derivation $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$.

Suppose $\mathtt{e}$ is a variable $\mathtt{x}$. Then $(\mathtt{x}{:}\tau) \in \Gamma$, and $\mathtt{q} = \mathtt{x}$. Therefore, $\theta(\mathtt{q}) = \mathtt{bools\ true\ true}$. Since $\mathtt{e}$ is a variable, it is normal and neutral. Therefore, (1) is true.

Suppose $\mathtt{e}$ is a λ abstraction $\lambda\mathtt{x}{:}\tau_1.\mathtt{e}_1$. Then $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$, and $\mathtt{q} = \mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1)$. The substitution $(\theta[\mathtt{x{:=}bools\ true\ true}])$ is of the required form for $\Gamma,(\mathtt{x}{:}\tau_1)$, so by induction, one of (1), (2), or (3) is true for $\Gamma,(\mathtt{x}{:}\tau_1) \vdash \mathtt{e}_1 : \tau_2 \rhd \mathtt{q}_1$ and $(\theta[\mathtt{x{:=}bools\ true\ true}])(\mathtt{q}_1)$. Now, we have that:

$$\begin{aligned}
&\theta(\mathtt{q})\\
&= \theta(\mathtt{abs}\ [\![\tau_1]\!]\ [\![\tau_2]\!]\ (\lambda\mathtt{x}{:}\mathtt{F}\ [\![\tau_1]\!].\ \mathtt{q}_1))\\
&= \mathtt{nfAbs}\ \theta([\![\tau_1]\!])\ \theta([\![\tau_2]\!])\ (\lambda\mathtt{x}{:}\mathtt{KBools}\ \theta([\![\tau_1]\!]).\ \theta(\mathtt{q}_1))\\
\longrightarrow^* &\ \mathtt{bools}\ (\mathtt{fst}\ (\theta(\mathtt{q}_1)[\mathtt{x{:=}bools\ true\ true}]))\ \mathtt{false})\\
&= \mathtt{bools}\ (\mathtt{fst}\ \theta(\mathtt{q}_1[\mathtt{x{:=}bools\ true\ true}]))\ \mathtt{false}\\
&= \mathtt{bools}\ (\mathtt{fst}\ (\theta[\mathtt{x{:=}bools\ true\ true}])(\mathtt{q}_1))\ \mathtt{false}
\end{aligned}$$

There are two cases: either $\mathtt{e}$ is normal and not neutral, or $\mathtt{e}$ is not normal.

Suppose e is normal and not neutral. Then e_1 is normal. Then (1) or (2) is true for e_1. Then either (θ[x:=bools true true])(q_1) $\longrightarrow^*$bools true true or (θ[x:=bools true true])(q_1) $\longrightarrow^*$bools true false. In either case fst (θ[x:=bools true true])(q_1) $\longrightarrow^*$true. Therefore,

θ(q)
$\longrightarrow^*$ bools (fst (θ[x:=bools true true])(q_1)) false
$\longrightarrow^*$ bools true false

So (2) is true for e.

Suppose e is not normal. By Lemma A.32, e_1 is not normal. Therefore, (θ[x:=bools true true])(q_1) $\longrightarrow^*$bools false false, and we have that

θ(q)
$\longrightarrow^*$ bools (fst (θ[x:=bools true true])(q_1)) false
$\longrightarrow^*$ bools (fst (bools false false)) false
$\longrightarrow^*$ bools false false

So (3) is true for e.

Suppose e is an application e_1 e_2. Then $\Gamma \vdash$ e_1 : $\tau_2 \rightarrow \tau \rhd$ q_1, and $\Gamma \vdash$ e_2 : $\tau_2 \rhd$ q_2, and q = app $[\![\tau_2]\!]$ $[\![\tau]\!]$ q_1 q_2. We have that

θ(q)
$=\theta$(app $[\![\tau_2]\!]$ $[\![\tau]\!]$ q_1 q_2)
$=$ nfApp $\theta([\![\tau_2]\!])$ $\theta([\![\tau]\!])$ $\theta(q_1)$ $\theta(q_2)$
$\longrightarrow^*$ bools (and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
(and (snd $\theta(q_1)$) (fst $\theta(q_2)$))

By induction, one of (1), (2) or (3) is true for each of e_1 and e_2 with θ. Suppose e is normal and neutral. Then e_1 is normal and neutral and e_2 is normal. Therefore, (1) is true for e_1 and either (1) or (2) is true for e_2. In particular, $\theta(q_1)$ $\longrightarrow^*$bools true true and either $\theta(q_2)$ $\longrightarrow^*$bools true true or $\theta(q_2)$ $\longrightarrow^*$bool true false. Therefore, fst $\theta(q_2)$ $\longrightarrow^*$true. Now, we have that

θ(q)
$\longrightarrow^*$ bools (and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
(and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
$\longrightarrow^*$ bools (and true true) (and true true)
$\longrightarrow^*$ bools true true

Therefore, (1) is true for e.

Suppose e is normal but not neutral. By Lemma A.32, e is neutral. Contradiction.

Suppose e is not normal. Then one of the following is true e_1 is not normal, e_1 is normal but not neutral, or e_2 is not normal. Suppose e_1 is not normal. Then (3) is true for e_1, so $\theta(q_1)$ $\longrightarrow^*$bools false false. Therefore,

θ(q)
$\longrightarrow^*$ bools (and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
(and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
$\longrightarrow^*$ bools (and false (fst $\theta(q_2)$))
(and false (fst $\theta(q_2)$))
$\longrightarrow^*$ bools false false

So (3) is true for e.

Suppose e_1 is normal but not neutral. Then (2) is true for e_1, so $\theta(q_1)$ $\longrightarrow^*$bools true false. Therefore θ(q) $\longrightarrow^*$bools false false similarly to the previous case, so (3) is true for e.

Suppose e_2 is not normal. Then (3) is true for e_2, so $\theta(q_2)$ $\longrightarrow^*$bools false false. Therefore,

θ(q)
$\longrightarrow^*$ bools (and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
(and (snd $\theta(q_1)$) (fst $\theta(q_2)$))
$\longrightarrow^*$ bools (and (snd $\theta(q_1)$) false)
(and (snd $\theta(q_1)$) false)
$\longrightarrow^*$ bools false false

So (3) is true for e.

Suppose e is a type abstraction $\Lambda\alpha{:}\kappa$.e_1. Then $\tau = (\forall\alpha{:}\kappa.\tau_1)$, and Γ,($\alpha{:}\kappa$) $\vdash$ e_1 : ($\forall\alpha{:}\kappa.\tau_1$) $\rhd$ q_1, and κ ► σ, and q = tabs $[\![\forall\alpha{:}\kappa.\tau_1]\!]$ $\text{strip}_{(\text{KBools},[\![\forall\alpha:\kappa.\tau 1]\!],[\![\tau 1]\!][\alpha:=\sigma],\sigma)}$ ($\Lambda\alpha{:}\kappa$.q_1). From now on we will abbreviate $\text{strip}_{(\text{KBools},[\![\forall\alpha:\kappa.\tau 1]\!],[\![\tau 1]\!][\alpha:=\sigma],\sigma)}$ as strip. By Lemma A.11, we have that (θ(strip) Bools ($\Lambda\alpha$:*.λx:Bools.x) ($\Lambda\alpha{:}\kappa$.$\theta(q_1)$))) $\longrightarrow^*$($\theta(q_1)$[α:=σ]). Therefore,

θ(q)
$=\theta$(tabs $[\![\forall\alpha{:}\kappa.\tau_1]\!]$ strip ($\Lambda\alpha{:}\kappa$.q_1))
$=$ nfTAbs $\theta([\![\forall\alpha{:}\kappa.\tau_1]\!])$ θ(strip) ($\Lambda\alpha{:}\kappa$.$\theta(q_1)$))
$\longrightarrow^*$ bools
(fst (θ(strip) Bools ($\Lambda\alpha$:*.λx:Bools.x)
($\Lambda\alpha{:}\kappa$.$\theta(q_1)$))))
false)
$=$ bools
(fst (strip Bools ($\Lambda\alpha$:*.λx:Bools.x)
($\Lambda\alpha{:}\kappa$.$\theta(q_1)$))))
false
$\longrightarrow^*$ bools (fst ($\theta(q_1)$[α:=σ])) false

By induction, one of (1), (2), or (3) is true for e_1 and θ. We have two cases: either e is normal and not neutral, or e is not normal. Suppose e is normal and not neutral. Then e_1 is normal, and either (1) or (2) is true for e_1 and θ. Then either $\theta(q_1)$[α:=σ] $\longrightarrow^*$(bools true true)[α:=σ] = bools true true, or $\theta(q_1)$[α:=σ] $\longrightarrow^*$(bools true false)[α:=σ] = bools true false. In either case, fst ($\theta(e_1)$[α:=σ]) $\longrightarrow^*$true. Therefore,

θ(q)
$\longrightarrow^*$ bools (fst ($\theta(q_1)$[α:=σ])) false
$\longrightarrow^*$ bools true false

So (2) is true for e.

Suppose e is not normal. Then by Lemma A.32, e_1 is not normal. Therefore, $\theta(q_1)$[α:=σ] $\longrightarrow^*$(bools false false)[α:=σ] = bools false false. Therefore,

θ(q)
$\longrightarrow^*$ bools (fst ($\theta(q_1)$[α:=σ])) false
$\longrightarrow^*$ bools (fst (bools false false)) false
$=$ bools false false

So (3) is true for e.

Suppose e is a type application e_1 σ. Then $\Gamma \vdash$ e_1 : ($\forall\alpha{:}\kappa.\tau_1$) $\rhd$ q_1, and τ = τ_1[α:=σ], and q = tapp $[\![\forall\alpha{:}\kappa.\tau_1]\!]$ q_1 $[\![\tau_1[\alpha:=\sigma]]\!]$ $\text{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],[\![\sigma]\!])}$. From now on, we abbreviate $\text{inst}_{([\![\forall\alpha:\kappa.\tau 1]\!],[\![\sigma]\!])}$ as inst. We have that

θ(q)
$=\theta$(tapp $[\![\forall\alpha{:}\kappa.\tau_1]\!]$ q_1 $[\![\tau_1[\alpha:=\sigma]]\!]$ inst)
$=$ nfTApp $\theta([\![\forall\alpha{:}\kappa.\tau_1]\!])$ $\theta(q_1)$ $\theta([\![\tau_1[\alpha:=\sigma]]\!])$ θ(inst)
$\longrightarrow^*$ bools (snd $\theta(q_1)$) (snd $\theta(q_1)$)

By induction, one of (1), (2), or (3) is true for e_1 and θ. Suppose e is normal and neutral. Then e is normal and neutral. Therefore, $\theta(q_1)$ $\longrightarrow^*$bools true true. Then we have that

θ(q)
$\longrightarrow^*$ bools (snd $\theta(q_1)$) (snd $\theta(q_1)$)
$\longrightarrow^*$ bools true true

So (1) is true for e.

Suppose e is normal and not neutral. By Lemma A.32, e is neutral. Contradiction. Therefore, it is false that e is normal and not neutral.

Suppose e is not normal. By Lemma A.32, e_1 is not normal and neutral. Therefore, either (2) or (3) is true for e_1, so either $\theta(q_1)$ $\longrightarrow^*$bools false false, or $\theta(q_1)$ $\longrightarrow^*$bools true false. In either case, snd $\theta(q_1)$ $\longrightarrow^*$false. Therefore, we have that:

θ(q)
$\longrightarrow^*$ bools (snd $\theta(q_1)$) (snd $\theta(q_1)$)
$\longrightarrow^*$ bools false false

So (3) is true for e. □

Theorem 7.10. *Suppose* $\langle\rangle \vdash$ e : τ.

1. *If* `e` *is* β*-normal, then* `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`true`*.*
2. *If* `e` *is not* β*-normal, then* `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`false`*.*

Proof. Suppose $\langle\rangle \vdash$ `e :` τ. Let `q` be such that $\langle\rangle \vdash$ `e :` τ $\rhd$ `q`. By Theorem 7.2, we have that `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`fst (q[F := KNat, abs := nfAbs, app := nfApp, tabs := nfTAbs, tapp := nfTApp])`. Let θ`=[F := KNat, abs := nfAbs, app := nfApp, tabs := nfTAbs, tapp := nfTApp]`, so `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`fst` θ`(q)`. Since $\langle\rangle$ is empty, θ is of the required form for lemma A.33.

We proceed by case analysis on `e`.

Suppose `e` is normal and neutral. By Lemma A.33, θ`(q)` $\longrightarrow^*$`bools true true`. Therefore, `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`fst` θ`(q)` $\longrightarrow^*$`true`.

Suppose `e` is normal and not neutral. By Lemma A.33, θ`(q)` $\longrightarrow^*$`bools true false`. Therefore, `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`fst` θ`(q)` $\longrightarrow^*$`true`.

Suppose `e` is not normal. By Lemma A.33, θ`(q)` $\longrightarrow^*$`bools false false`. Therefore, `nf` $\overline{\tau}$ $\overline{\texttt{e}}$ $\longrightarrow^*$`fst` θ`(q)` $\longrightarrow^*$`false`.

$\square$