Self-Representation in Girard's System U

Matt Brown
UCLA
msb@cs.ucla.edu

Jens Palsberg
UCLA
palsberg@ucla.edu

Abstract

In 1991, Pfenning and Lee studied whether System F could support a typed self-interpreter. They concluded that typed self-representation for System F "seems to be impossible", but were able to represent System F in F_ω. Further, they found that the representation of F_ω requires kind polymorphism, which is outside F_ω. In 2009, Rendel, Ostermann and Hofer conjectured that the representation of kind-polymorphic terms would require another, higher form of polymorphism. Is this a case of infinite regress? We show that it is not and present a typed self-representation for Girard's System U, the first for a λ-calculus with decidable type checking. System U extends System F_ω with kind polymorphic terms and types. We show that kind polymorphic types (i.e. types that depend on kinds) are sufficient to "tie the knot" – they enable representations of kind polymorphic terms without introducing another form of polymorphism. Our self-representation supports operations that iterate over a term, each of which can be applied to a representation of itself. We present three typed self-applicable operations: a self-interpreter that recovers a term from its representation, a predicate that tests the intensional structure of a term, and a typed continuation-passing-style (CPS) transformation – the first typed self-applicable CPS transformation. Our techniques could have applications from verifiably type-preserving metaprograms, to growable typed languages, to more efficient self-interpreters.

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; Types

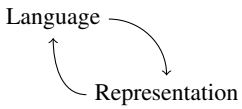
1. Introduction

Typed self-representation is the problem of representing a statically typed language in itself. It can be seen as the intersection of two lines of research: self-representation, which generally studies representations of untyped or dynamically typed languages in themselves, and typed representation, which studies techniques for defining typed representations of statically typed languages that ensure only well-typed programs can be represented.

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676988

In general, the techniques required for building a representation depend upon both the meta-language (in which the representation is defined) and the object language (which is represented). Representations of expressive object language features tend to require even more expressive meta-language features. In the case of a self-interpreter, the meta-language and the object language are the same. The key challenge of typed self-representation is to identify a single typed language that is expressive enough to represent each of its own features, without additional expressive power.



In our case, we are interested in typed λ-calculi with decidable type checking. It has been an open question since 1991 [26] whether a typed λ-calculus with decidable type checking can support a meaningful notion of typed self-representation.

Self Representation. Self-representation and self-interpretation have many important applications. A self-interpreter can be used to grow a language from a small core implemented in some other meta-language. One can use similar techniques to implement self-optimizers and compilers, as well as debuggers, read-eval-print-loops, and macro systems. A similar idea is the reflective tower, which uses an infinite tower of self-interpreters to add reflective capabilities to a language.

There are many examples of self-interpreters in the literature, including ones for λ-calculus [4, 7, 8, 18, 22, 23, 28, 31], Haskell [25], JavaScript [13], Lisp [21], Python [36], Ruby [37], Scheme [2], Standard ML [29], and many others [19, 32, 38]. In each of these the representations are untyped, in the sense that (1) it is possible to build representations of ill-typed terms, and (2) all representations are either untyped or else have the same type.

Typed Representation. We can contrast this with typed representations, which have two essential properties: (1) only well-typed terms can be represented, and (2) the type of a term is reflected in the type of its representation, in the sense that the former can be determined by the latter. This provides important correctness guarantees for metaprograms. An immediate consequence of (1) is that a metaprogram cannot produce ill-typed terms. We can also ensure that the types of its input and output terms are related in a particular way. For example, we can ensure that a self-interpreter preserves the type of its input, or that a continuation-passing-style transformation modifies the type of the input program in the expected way.

There are many examples in the literature of typed representations. In most cases the techniques rely on the fact that the meta-language has a more powerful type system than the object language.

Typed Self-Representation. The goal of typed self-representation is to combine the benefits of self-representation and typed representation. It promises the best of both worlds: on the one

hand, it brings the expressive power of self-representation to the world of statically typed languages. On the other hand, it brings the correctness guarantees of types to self-applicable metaprograms. A robust typed self-representation would support typed variants of the kinds of applications enabled by self-representation. It would also narrow the expressiveness gap between dynamically typed languages and statically typed languages, allowing more classical programs from dynamically typed languages to be statically type checked.

What does it mean for a language to support typed self-representation? We have adopted two primary requirements: that our language be a typed λ-calculus with decidable type-checking, and that a representation support operations that iterate over the structure of the term. Further, we want to allow operations that produce results of different types, possibly related to the type of the input representation.

We target a typed self-recognizer [17], which is a self-interpreter that recovers a term from its representation. The idea of a self-recognizer was first studied by Kleene [18] in 1936 for an untyped λ-calculus. There are several examples of typed recognizers and self-recognizers [26, 27] that are implemented by iteration. Iteration is desirable because it can be supported by languages that don't include recursion. An operation that iterates or folds over the term is defined by cases – one case for each syntactic form.

In the case of a pure λ-calculus (that contains only abstractions, applications, and variables), we have identified a core challenge that is related to typed self-representation. The Polymorphic Application (polyapp) problem is to define a polymorphic application function for each form of application in the language. For example, System F terms can be applied to terms and to types. The polyapp problem for System F is to define a polymorphic application function that can apply terms to terms, and a polymorphic application function that can apply terms to types. In Section 3.2 we present a general technique for implementing polyapp functions by decomposing types. In subsequent sections we leverage this technique to represent terms and types.

History. Typed self-representation of λ-calculi has been studied since at least 1991, when Pfenning and Lee [26] considered whether System F could support a typed self-recognizer. Pfenning and Lee concluded that the problem "seems to be impossible", but were able to implement a typed recognizer for System F representations in F_ω. Furthermore, they were able to implement F_ω in F_ω^+, which extends F_ω with kind polymorphic terms. They did not study representation of F_ω^+.

In 2009 Rendel, Ostermann and Hofer [27] studied the representation of kind polymorphism, and conjectured that it would require "another, higher form of polymorphism". Their solution was to combine the categories of types and kinds, so that kind polymorphism is represented in the same way as type polymorphism. They demonstrated the first typed self-representation and first typed self-recognizer for a λ-calculus, though their calculus does not support decidable type checking.

In 2011 Jay and Palsberg [17] implemented a typed self-interpreter for a combinator calculus with System F types. They implemented a self-recognizer and the first typed self-enactor, a self-interpreter that implements multi-step reduction on typed representations. Like [27], type checking is undecidable in their calculus. Their representation technique was designed for combinators, and does not appear to be easily translated to a λ-calculus.

Tying the Knot. A challenge of representing a typed λ-calculus is to find techniques for representing each form of abstraction and application in the language without adding any new ones. Pfenning and Lee represented System F type abstraction and application using the higher order types of F_ω. In particular, they used higher order types to represent System F type abstractions and applications.

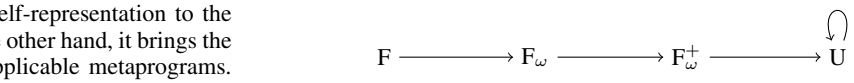


Figure 1. Four typed λ-calculi: $\rightarrow$ denotes "represented in."

We can demonstrate their technique using a polymorphic type application function.

A type application function for System F type applications should map a polymorphic term and a type to the application of the term to the type. For example, given a term of type $\forall\alpha.\ \alpha \to \alpha$ and a type τ, the type application function should return a term of type $\tau \to \tau$. We can define the type application function for this case as $\lambda x : (\forall\alpha.\ \alpha \to \alpha).\ \Lambda\beta.\ x\ \beta$.

A polymorphic type application function for System F should be able to apply any polymorphic term. In other words, it should be polymorphic in the type of the term, and the type of the term must itself be polymorphic. What is needed is a way to abstract over exactly the polymorphic types of System F, excluding monomorphic types like $\alpha \to \beta$. This is beyond the capabilities of System F type abstraction.

Pfenning and Lee solved this problem by encoding quantified types as second-order types in System F_ω. For example, the type $\forall\alpha.\ \alpha \to \alpha$ can be encoded as the second-order type $\lambda\alpha : *.\ \alpha \to \alpha$. The types of F_ω are classified by a family of kinds. First-order types like $\tau_1 \to \tau_2$ have kind $*$, and second-order types like $\lambda\alpha : *.\ \alpha \to \alpha$ have kind $* \to *$. Type abstractions in F_ω range over a particular kind that is specified by an annotation. We can abstract over encodings of System F quantified types using a type abstraction annotated with kind $* \to *$. This enables a polymorphic type application function for System F quantified types to be implemented as:

$$\Lambda\sigma : * \to *.\ \lambda x : (\forall\alpha : *.\ \sigma\ \alpha).\ \Lambda\beta : *.\ x\ \beta$$

In F_ω, the type of this term is $\forall\sigma : * \to *.\ (\forall\alpha : *.\ \sigma\ \alpha) \to (\forall\beta : *.\ \sigma\ \beta)$. The type $\forall\alpha : *.\ \sigma\ \alpha$ represents an arbitrary quantified type. Substituting σ with an encoded quantified type recovers the quantified type. For example, substituting σ with $\lambda\alpha : *.\ \alpha \to \alpha$ yields $\forall\alpha : *.\ (\lambda\alpha : *.\ \alpha \to \alpha)\ \alpha$, which is equivalent to $\forall\alpha : *.\ \alpha \to \alpha$. Note that $\forall\alpha : *.\ \alpha \to \alpha$ is the F_ω version of $\forall\alpha.\ \alpha \to \alpha$. Since every System F quantified type can be encoded as an F_ω type of kind $* \to *$, this type application function can apply any polymorphic System F term. It can only apply polymorphic terms, because no substitution for σ can make $(\forall\alpha : *.\ \sigma\ \alpha)$ into a monomorphic type.

Pfenning and Lee used this technique to represent System F in F_ω. They were unable to represent System F_ω in itself, but did achieve a representation of F_ω in F_ω^+, which extends F_ω with kind abstraction and application in terms. It is easy to imagine that this is a case of infinite regress; that representing kind-abstractions will require another extension, which will also need to be represented if we hope to achieve self-representation.

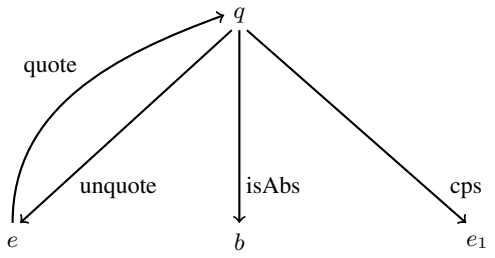
The System F type application function discussed above already hints at the question of infinite regress. It can apply any polymorphic term typeable in System F, but is not itself typeable in System F. On the other hand, it is typeable in System F_ω, but can only apply some of the polymorphic terms in F_ω. In particular, it cannot apply itself. This begs the following question, which we name the Polymorphic Application problem: is it possible to define a set of polymorphic application functions in a particular language, one for each form of application in the terms of that language (e.g., for applications of terms to terms, terms to types, terms to kinds, etc.)? We conjecture that a language that supports typed self-representation can also support polymorphic application.

In Section 3.2 we formalize the Polymorphic Application problem and present a solution for Girard's System U. In later sections we use our solution to define our typed self-representation for System U. Our result is summarized in Figure 1. Pfenning and Lee were able to represent System F in F_ω, and F_ω in F^+_ω. We show that F^+_ω can be represented in System U, and that System U can represent itself.

System U. System U was first introduced by Jean-Yves Girard in his PhD thesis [16], in which he also introduced System F. Girard used System U to formalize a version of the Burali-Forti paradox. Girard's paradox showed that System U is not strongly normalizing, and that every type is inhabited. Thus, as a logic, System U is inconsistent.

System U is a Pure Type System that lies outside the λ-cube [5], and that does not include dependent types. It is an extension of F^+_ω, and every legal F^+_ω term is a legal System U term. The terms of System U consist of variables and abstractions and applications of each of terms, types, and kinds. The types of System U consist of variables and abstractions and applications of each of types and kinds. Intuitively, the types of System U are the terms of System F. In Section 3.2 we show that System U can support a polymorphic application function for each form of application. A key property that makes tying the knot possible is that System U does not have higher-order kinds. As a result, there is no "type system" for kinds: all kinds are classified by a single sort $\Box$. This is analogous to the types of System F: System F does not have higher-order types, and all types are classified by a single sort $*$. Since System U has only one classifier of kinds, a representation of System U does not need to abstract over classifiers of kinds.

Representation and Operations. We represent both terms and the types of terms. We call our term representation procedure "quotation". In section 5, we define a meta-level process of quotation, which formalizes what it means for one term to represent another. In the diagram above, our quotation function quote maps a typed term to its representation. Unlike the other operations in the figure, quote is defined outside the language itself.



We support operations that iterate or fold over the representation. In section 6 we present three example operations defined as folds: a self-recognizer unquote that recovers a term from its representation, a predicate isAbs that tests the intensional structure of a representation, and a typed continuation-passing style (CPS) transformation. CPS transformation is often used in compilers of functional programming languages.

Our Results. We identify System U as the first known typed λ-calculus with decidable type checking that supports typed self-representation. We represent both terms and the types of terms, which enables operations that transform the type of their input. Type representations are essential to our implementation of the first typed self-applicable CPS transformation. The result type of the CPS transformation is a function of the intensional structure of the input type.

Our type representations are types of a particular kind U. Type representations are used to type check term representations. For example, suppose that e is a term of type τ, that q is the representation of e, and that σ is the representation of τ. Then the type of q is Exp σ. Our self-recognizer unquote recovers a term from its representation, so that unquote σ q $\equiv_\beta$ e. The type of unquote is $\Pi\alpha$: U. Exp $\alpha \to$ UId α. In Pure Type Systems, Π is analogous to the $\forall$ quantifier of System F and F_ω. The type UId is an operation on type representations that recovers a type from its representation. For example UId $\sigma \equiv_\beta \tau$.

Rest of the Paper. Section 2 gives an overview of Pure Type Systems, Section 3 describes System U, Section 4 defines our representation of types, Section 5 defines our representation of terms, Section 6 presents our example operations, Section 7 discusses our implementation and experimental results, Section 8 contains a comparison with related work, Section 9 discusses future work, and Section 10 concludes. Proofs of theorems stated throughout the paper are provided in the appendix of the full paper, which is available from our website [1].

2. Pure type systems

We use Barendregt's [5] formalization of System U as a Pure Type System (PTS). This section gives an overview of some important aspects of PTSs for the unfamiliar reader, but does not include a detailed tutorial. Pure Type Systems have a uniform syntax, which helps to clarify both the presentation of our self-representation techniques, and a comparison between System U and other PTS instances like System F and F_ω. The comparison serves to explain what parts of System U are important for achieving a self-representation.

A Pure Type System is defined by a set of expressions $\mathcal{T}$ and a specification. The expressions are defined by the grammar:

$$\mathcal{T} = V \mid C \mid \lambda V : \mathcal{T}. \mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \Pi V : \mathcal{T}. \mathcal{T}$$

Here V ranges over a countable set of variables and C ranges over a set of constants. We use x, y to range over variables, c to range over constants, and a, b, A, and B to range over expressions. Functionals are introduced by the λ form and eliminated by application. The form $\Pi V : \mathcal{T}. \mathcal{T}$ introduces a product, which is used to classify functionals.

The notation A $\to_\beta$ B denotes that A reduces to B in one step of β-reduction. Similarly, A $\to_\eta$ B denotes that A reduces to B in one step of η-reduction, and A $\to_{\beta\eta}$ B denotes that either A $\to_\beta$ B or A $\to_\eta$ B. The relation $\twoheadrightarrow_\beta$ denotes the reflexive transitive closure of $\to_\beta$, and $\equiv_\beta$ denotes the least congruence relation generated by $\to_\beta$. The relations $\twoheadrightarrow_\eta$, $\equiv_\eta$, $\twoheadrightarrow_{\beta\eta}$, and $\equiv_{\beta\eta}$ are defined similarly.

A specification of a PTS consists of a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$. The first component $\mathcal{S}$ is a subset of C called sorts. We will use s to range over sorts. In the systems we consider, all constants are sorts. The second component $\mathcal{A}$ is a set of axioms of the form $c : s$, where c is a constant and s a sort. The third component $\mathcal{R}$ is a set of rules of the form (s_1, s_2, s_3), for some sorts s_1, s_2, and s_3. We use the shorthand (s_1, s_2) to denote (s_1, s_2, s_2). The specification and a set of derivation rules determine the derivable typing judgments $\Gamma \vdash$ A : B.

In a judgment of the form $\Gamma \vdash$ A : B, we call A the *subject* and B the *classifier* of A. If $\Gamma \vdash$ A : B can be derived using the rules in Figure 2 with the specification of a particular PTS, then A is *legal* in that system. Some authors call the set of legal expressions the *terms*, though we will use *term* to refer to a subset of the legal expressions defined below.

We call a product derived with the rule (s_1, s_2, s_3) as its side-condition an "(s_1, s_2, s_3) product". The rule ensures that a (s_1, s_2, s_3) product will be classified by s_3. A product Πx : A. B is called a dependent product if x occurs free in B. It is standard to abbreviate products Πx : A. B as A $\to$ B when x does not occur free in B. It is sometimes possible to determine that an arbitrary (s_1, s_2, s_3) product can be written in this abbreviated form. For

$$\text{axioms}\ \frac{}{\vdash \mathrm{c} : \mathrm{s}}\ (\mathrm{c{:}s}) \in \mathrm{A}$$

$$\text{start}\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

$$\text{weakening}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s}{\Gamma, x : B \vdash a : A}$$

$$\text{product}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3}\ (s_1, s_2, s_3) \in R$$

$$\text{abstraction}\ \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$$

$$\text{application}\ \frac{\Gamma \vdash F : (\Pi x : A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[x := a]}$$

$$\text{conversion}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B \equiv_\beta B'}{\Gamma \vdash A : B'}$$

Figure 2. The rules for a PTS $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$

example, in System F, a $(*,*)$ product has the form Πx : A. B, where x ranges over terms and A and B are types. Since System F does not include dependent types, x cannot occur free in B, so Πx : A. B can be written A $\rightarrow$ B.

To demonstrate the uniformity of PTS syntax, consider the System F term $\Lambda\alpha.\ \lambda$x : α. x which has the System F type $\forall\alpha.\alpha \rightarrow \alpha$. The symbol Λ denotes a type abstraction. The type variable α is not classified, since System F does not classify types. The symbol λ denotes a term abstraction, and the term variable x is classified. The universal quantifier $\forall$ forms the types of Λ abstractions, and $\rightarrow$ forms the types of λ abstractions. In PTS syntax, the term is written $\lambda\alpha : *.\ \lambda$x : α. x, and the type is written $\Pi\alpha : *.\ \Pi$x : $\alpha.\ \alpha$. Here the same symbol λ is used for the both abstractions, and the type of each λ abstraction is a product. Since x does not occur free in α, we can also write the type as $\Pi\alpha : *.\ \alpha \rightarrow \alpha$.

Theorem 2.1 (Subject Reduction for $\rightarrow_{\beta\eta}$ [15])**.** *If* $\Gamma \vdash$ A : B *and* A $\twoheadrightarrow_{\beta\eta}$ A$'$*, then* $\Gamma \vdash$ A$'$: B

Theorem 2.2 (Church-Rosser for $\rightarrow_\beta$ [15])**.** *If* $\Gamma \vdash$ A : B *and* A $\twoheadrightarrow_\beta$ A_1 *and* A $\twoheadrightarrow_\beta$ A_2*, then there exists* A$'$ *such that* $\mathrm{A}_1 \twoheadrightarrow_\beta$ A$'$ *and* $\mathrm{A}_2 \twoheadrightarrow_\beta$ A$'$.

While the focus of this paper is primarily on System U, we also discuss Systems F, F_ω and F^+_ω. The PTS specification of each system is listed in Figure 3. The sorts of each is a subset of $\{*, \Box, \Delta\}$. We divide the legal expressions of each PTS based on these sorts: the sort $*$ corresponds to the *terms*, the sort $\Box$ corresponds to the *types*, and the sort Δ corresponds to the *kinds*. More precisely, suppose a legal expression A is derived by $\Gamma \vdash A : B$. If $\Gamma \vdash B : *$, then A is a term. If $\Gamma \vdash B : \Box$, then A is a type. If $\Gamma \vdash B : \Delta$, then A is a kind.

The axioms $\mathcal{A}$ of a PTS instance defines how the different sorts are related to each other. The axioms of each PTS instance listed in Figure 3 are a subset of $\{* : \Box, \Box : \Delta\}$. The axiom $(* : \Box)$ means that $\Box$ is the classifier of $*$ and that types classify terms. The axiom $(\Box : \Delta)$ means that Δ is the classifier of $\Box$ and that the kinds classify types. We call the classifier of a term its type, and the classifier of a type its kind.

The rules $\mathcal{R}$ determine which products are legal in a PTS instance. The rules $\mathcal{R}$ of each PTS instance listed in Figure 3 are a

System F	$\mathcal{S}$	$*, \Box$
	$\mathcal{A}$	$* : \Box$
	$\mathcal{R}$	$(*,*), (\Box,*)$
System F_ω	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*,*), (\Box,*), (\Box,\Box)$
System F^+_ω	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*,*), (\Box,*), (\Box,\Box), (\Delta,*)$
System U	$\mathcal{S}$	$*, \Box, \Delta$
	$\mathcal{A}$	$* : \Box, \Box : \Delta$
	$\mathcal{R}$	$(*,*), (\Box,*), (\Box,\Box), (\Delta,*), (\Delta,\Box)$

Figure 3. PTS specifications of key calculi

subset of $\{(*,*), (\Box,*), (\Box,\Box), (\Delta,*), (\Delta,\Box)\}$. The rule $(*,*)$ derives terms that abstract over other terms, the types of which are $(*,*)$ products. The rule $(\Box,*)$ derives terms that abstract over types, the types of which are $(\Box,*)$ products. The rule $(\Box,\Box)$ derives types that abstract over other types, the kinds of which are $(\Box,\Box)$ products. The rule $(\Delta,*)$ derives terms that abstract over kinds, the types of which are $(\Delta,*)$ products. The rule $(\Delta,\Box)$ derives types that abstract over kinds, the kinds of which are $(\Delta,\Box)$ products.

In each PTS of Figure 3, legal products are either types or kinds. A product type is a legal product that is a type. It is necessarily a $(s,*)$ product. A product kind is a legal product that is a kind. It is necessarily a $(s,\Box)$ product.

We adopt a naming convention to help distinguish between terms, types, and kinds. Names starting with lower-case letters such as e refer to terms. Names starting with upper-case letters and the greek letters α, β, τ, and σ refer to types. Greek letters κ and χ refer to kinds.

Systems F and F_ω are part of the λ-cube [5], while F^+_ω and U are not because of the rules $(\Delta,*)$ and $(\Delta,\Box)$. All except System U are strongly normalizing. Girard's thesis [16] proved that System U is not strongly normalizing. In particular, there exists a legal term in System U with the type $\Pi\alpha : *.\alpha$ that does not have a normal form. However, the types and kinds of System U are strongly normalizing.

Type checking is decidable for each of System F, F_ω, F^+_ω, and U [6]. This relies on that each system is *injective* [14], a technical property of Pure Type Systems. An injective PTS also has the property that types are unique [5].

None of the PTS instances shown in Figure 3 have dependent types (e.g., types that abstract over terms), which require the rule $(*,\Box)$. A consequence of this is that any $(*,*)$ product $\Pi x : \tau_1.\ \tau_2$ can be written $\tau_1 \rightarrow \tau_2$. Types cannot depend on terms, so x cannot occur free in τ_2.

3. System U

In this section we define a decomposition of System U product types and use it to solve the Polymorphic Application problem for System U. The ideas in this section recur throughout the paper: in the definition of type representations in Section 4; in the decomposition of type representations in Section 4; and in the definition of term representations in Section 5.

3.1 Decomposing Product Types

In a pure type system, products are always classified by a sort. The sort of types is $*$, so product types are products classified by $*$.

The product types of System U are formed by $(*,*)$, $(\Box,*)$, and $(\Delta,*)$. In general, if a product $\Pi x : A.\ \tau$ is formed by $(s,*)$, then τ is classified by $*$ (i.e. τ is a type) and A is classified by s. System U products formed by $(*,*)$ and by $(\Delta,*)$ are special cases. A $(*,*)$ product of the form $\Pi x : \tau_1.\tau_2$ is special because the bound variable x cannot occur free in τ_2 (i.e. types cannot depend on terms). Products formed by $(\Delta,*)$ are special because the only subject classified by Δ is $\Box$.

Theorem 3.1. *In System U, if* $\Gamma \vdash A : \Delta$, *then* $A = \Box$.

Proof. By induction on the height of the derivation. Assume $\Gamma \vdash A : \Delta$. We proceed by considering the last rule in the derivation. If the last rule is axioms, then $(A : \Delta) \in \mathcal{A}$. The only possibility is $(\Box, \Delta)$, so $A = \Box$ as required. If the last rule is start, then $(\Delta : s) \in \mathcal{A}$. Contradiction. If the last rule is weakening, then $\Gamma = \Gamma_1, x : C$ and $\Gamma_1 \vdash A : \Delta$. By induction, $A = \Box$ as required. If the last rule is product, then there exist sorts s_1, s_2 such that $(s_1, s_2, \Delta) \in \mathcal{R}$. Contradiction. If the last rule is abstraction, then $\Delta = \Pi x : A.B$. Contradiction. If the last rule is application, then $B[x := a] = \Delta$. There are two cases: either $B = \Delta$, or $a = \Delta$. If $B = \Delta$, then $(\Pi x : A.\Delta)$ must be legal. This in turn requires a sort s and an axiom $\Delta : s$. Contradiction. If $a = \Delta$, then there must exist a term A such that $\Gamma \vdash \Delta : A$. Contradiction. If derived by conversion, then there must exist a sort s such that $\Gamma \vdash \Delta : s$, which in turn requires an axiom $\Delta : s$. Contradiction. □

In words, Theorem 3.1 states that System U does not include a "type system" for kinds. The situation is similar for the types of System F, as it does not include a kind system (which is a "type system" for types). In the PTS formulation of System F, the only subject classified by $\Box$ is $*$. As we will see, Theorem 3.1 is a key property of System U that enables self-representation.

Theorem 3.2. *If* $\Gamma \vdash \tau : *$, *and* τ *is a normal form, then* τ *is of one of the following forms:*

$\alpha\ A_1\ \ldots\ A_n$	where α is a type variable.
$\tau_1 \to \tau_2$	where $\Gamma \vdash \tau_1 : *$
$\Pi\alpha : \kappa.\ \tau_1$	where $\Gamma \vdash \kappa : \Box$
$\Pi\chi : \Box.\ \tau_1$	

Since we are only interested in decomposing product types (and not applications of the form $\alpha\ \tau_1\ \ldots\ \tau_n$), we only consider the last three cases. We begin by defining a constructor for each case of product, corresponding to the rule $(s,*)$ that forms the product.

Definition 3.1 (Constructors for product types)**.** *We define the following constructors for product types:*

$$\begin{aligned}\pi_* &= \lambda\alpha : *.\ \lambda\beta : *.\ \alpha \to \beta\\ \pi_\Box &= \lambda\chi : \Box.\ \lambda\alpha : \chi \to *.\Pi\beta : \chi.\ \alpha\ \beta\\ \pi_\Delta &= \lambda\alpha : \Box \to *.\Pi\chi : \Box.\ \alpha\ \chi\end{aligned}$$

It is straightforward to check that the product type constructors have the types given by the following judgments:

$$\begin{aligned}\langle\rangle &\vdash \pi_* : * \to * \to *\\ \langle\rangle &\vdash \pi_\Box : \Pi\chi : \Box.\ (\chi \to *) \to *\\ \langle\rangle &\vdash \pi_\Delta : (\Box \to *) \to *\end{aligned}$$

Every product type formed by $(s,*)$ can be equivalently expressed as an application of the constructor π_s. This is akin to a higher order abstract syntax encoding of product types [30].

Theorem 3.3 (Decomposition of product types)**.** *For any legal* $(*,*)$ *product* $\tau_1 \to \tau_2$, *any legal* $(\Box,*)$ *product* $\Pi\alpha : \kappa.\ \tau$, *and any legal* $(\Delta,*)$ *product* $\Pi\chi : \Box.\ \tau$, *we have:*

$$\begin{aligned}\tau_1 \to \tau_2 &\equiv_\beta \pi_*\ \tau_1\ \tau_2\\ \Pi\alpha : \kappa.\ \tau &\equiv_\beta \pi_\Box\ \kappa\ (\lambda\alpha : \kappa.\ \tau)\\ \Pi\chi : \Box.\ \tau &\equiv_\beta \pi_\Delta\ (\lambda\chi : \Box.\ \tau)\end{aligned}$$

Below we define the components of a product to be the arguments of the constructor that yield an equivalent type.

Definition 3.2 (Components of products)**.**

- *The components of a* $(*,*)$ *product type* $\tau_1 \to \tau_2$ *are* τ_1 *and* τ_2.
- *The components of a* $(\Box,*)$ *product type* $\Pi\alpha : \kappa.\ \tau$ *are* κ *and* $\lambda\alpha : \kappa.\ \tau$.
- *The component of a* $(\Delta,*)$ *product type* $\Pi\chi : \Box.\ \tau$ *is* $\lambda\chi : \Box.\ \tau$.

The following theorem states that the components of a product are always legal in the same environment as the product itself.

Theorem 3.4 (Types of product components)**.**

1. *If* $\Gamma \vdash (\Pi x : \tau_1.\ \tau_2) : *$ *and* $\Gamma \vdash \tau_1 : *$, *then* $\Gamma \vdash \tau_2 : *$.
2. *If* $\Gamma \vdash (\Pi\alpha : \kappa.\ \tau) : *$ *and* $\Gamma \vdash \kappa : \Box$, *then* $\Gamma \vdash (\lambda\alpha : \kappa.\ \tau) : \kappa \to *$.
3. *If* $\Gamma \vdash (\Pi\chi : \Box.\ \tau) : *$, *then* $\Gamma \vdash (\lambda\chi : \Box.\ \tau) : \Box \to *$.

As stated earlier, the key properties of System U that enable our self-representation technique are that product types can be decomposed, and that terms and types can abstract over the components. The components of a $(*,*)$ product are two types, and can be abstracted in terms and types by the rules $(\Box,*)$ and $(\Box,\Box)$, respectively. The components of a $(\Box,*)$ product are a kind and a type, and can be abstracted in terms by the rules $(\Delta,*)$ and $(\Box,*)$, and in types by $(\Delta,\Box)$ and $(\Box,\Box)$. The single component of a $(\Delta,*)$ component is a type, and can be abstracted over in terms and types by $(\Box,*)$ and $(\Box,\Box)$. These properties will play key roles in our solution of the Polymorphic Application problem (Section 3.2), and in the representation of types (Section 4) and terms (Section 5).

3.2 Polymorphic Application

We can now formalize the requirements of a polymorphic application functions and state the Polymorphic Application problem for a class of Pure Type Systems.

Definition 3.3 (Standard PTS)**.** *A PTS* $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ *is* standard *if:*

- *The designated sort* $* \in \mathcal{S}$ *classifies the types of terms.*
- *For each sort* $s \in S$, $(s : *) \notin \mathcal{A}$.
- *If* $(s_1, s_2, *) \in \mathcal{R}$, *then* $s_2 = *$.

The first condition of a Standard PTS establishes $*$ as the sort corresponding to terms. The second condition states that terms do not classify anything. The third condition states that if a term is an abstraction, then its body is also a term. The systems listed in Figure 3 and those in the λ-cube are all standard.

Definition 3.4 (Polymorphic Application Function)**.** *Let* $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ *be a Standard PTS, let* $(s,*) \in \mathcal{R}$, *and let* p *be a legal closed term. We say that* p *is a polymorphic application function for the* $(s,*)$ *products of* $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ *if it satisfies the following two conditions:*

- p *is of the form*

  $$\lambda x_1 : A_1.\ \ldots\ \lambda x_n : A_n.\ \lambda x : (\Pi b_1 : B.\tau).\ \lambda b_2 : B.\ x\ b_2$$

  for some $x_1,\ \ldots\ x_n, A_1,\ \ldots\ A_n, b_1, b_2, B, \tau$ *such that*

  $$x_1 : A_1, \ldots, x_n : A_n \vdash B : s.$$

- *For every closed* $(s,*)$ *product* σ *of* $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$, *there exist legal expressions* $a_1, \ldots, a_n$ *such that*

  $$\langle\rangle \vdash p\ a_1\ \ldots\ a_n : \sigma \to \sigma$$

The first condition defines the form of a polymorphic application function for $(s,*)$ products. The n outermost abstractions are what make it polymorphic: they abstract over the component(s) of such products. The term under the n outermost abstractions,

$\lambda x : (\Pi b_1 : B.\, \tau).\, \lambda b_2 : B.\, x\, b_2$, should be an application function for an arbitrary $(s, *)$ product. The second condition states that we can obtain an application function for particular closed $(s, *)$ product by n applications.

Definition 3.5 (Polymorphic Application Problem)**.** *For a Standard PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$, we say that $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ supports polymorphic application if there exists a legal polymorphic application function* polyapp$_s$ *for every rule $(s, *) \in \mathcal{R}$.*

For example, a solution of the Polymorphic Application problem for System F requires two polyapp functions that are legal System F terms: a polyapp$_*$ function to apply terms to terms, and a polyapp$_\Box$ function to apply terms to types. We conjecture that no legal polyapp$_\Box$ function exists for System F, and therefore that the Polymorphic Application problem for System F is impossible.

We now show how to solve the Polymorphic Application problem for System U. The solution consists of three application functions: polyapp$_*$, polyapp$_\Box$, and polyapp$_\Delta$. Later we will use the techniques from this section to define our representations of types and terms.

Term applications. Our first polymorphic application function polyapp$_*$ applies terms to terms. Terms that can be applied to other terms have $(*, *)$ product types of the form $\tau_1 \to \tau_2$, where $\langle\rangle \vdash \tau_1 : *$. An application function for terms of type $\tau_1 \to \tau_2$ will have the type $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$, and can be implemented as:

$$\lambda \text{f} : \tau_1 \to \tau_2.\, \lambda \text{a} : \tau_1.\, \text{f a}$$

By Theorem 3.4, we have that $\langle\rangle \vdash \tau_2 : *$. Therefore, we can make this polymorphic by abstracting out τ_1 and τ_2. The resulting function, polyapp$_*$, implements application of $(*, *)$ functions:

$$\text{polyapp}_* = \lambda \tau_1 : *.\, \lambda \tau_2 : *.\, \lambda \text{f} : \tau_1 \to \tau_2.\, \lambda \text{a} : \tau_1.\, \text{f a}$$

The abstractions of τ_1 and τ_2 are type abstractions formed by $(\Box, *)$. The type of polyapp$_*$ is:

$$\Pi \tau_1 : *.\, \Pi \tau_2 : *.\, (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$$

Lemma 3.1. polyapp$_*$ *is a polymorphic application function for $(*, *)$ products in System U.*

Proof. We have already seen that polyapp$_*$ is legal. It is easily checked that it has the required form, and that $\tau_1 : *, \tau_2 : * \vdash \tau_1 : *$. Let $\sigma_1 \to \sigma_2$ be closed a $(*, *)$ product in System U. By Theorem 3.4, the components σ_1 and σ_2 are both closed types of kind $*$. Therefore polyapp$_*$ σ_1 σ_2 : $(\sigma_1 \to \sigma_2) \to \sigma_1 \to \sigma_2$ as required. □

Type applications. Our second polymorphic application function polyapp$_\Box$ applies terms to types. Terms that can be applied to types have $(\Box, *)$ product types of the form $\Pi \alpha : \kappa.\, \tau$, where α ranges over types of kind κ and τ is a type. Based on the type rules in Figure 2, an application function for terms of a particular $(\Box, *)$ product type $\Pi \alpha : \kappa.\, \tau$ should have the type:

$$(\Pi \alpha : \kappa. \tau) \to \Pi \beta : \kappa.\, (\tau[\alpha := \beta]) \qquad (\dagger)$$

Here $\tau[\alpha := \beta]$ denotes the type obtained by substituting β for α in τ. The syntax $\tau[\alpha := \beta]$ is not part of our language of types, but can be expressed as $(\lambda \alpha : \kappa.\, \tau)\, \beta$. Letting τ' denote $\lambda \alpha : \kappa.\, \tau$, we can write $(\dagger)$ as:

$$(\Pi \alpha : \kappa.\, \tau'\, \alpha) \to \Pi \beta : \kappa.\, \tau'\, \beta$$

We can implement a polymorphic application function polyapp$_\Box$ for $(\Box, *)$ products by abstracting over κ and τ', which are the components of $\Pi \alpha : \kappa.\, \tau$. Theorem 3.4 states that κ has sort $\Box$ and τ' has kind $\Box \to *$.

$$\text{polyapp}_\Box = \lambda \kappa : \Box. \lambda \tau' : \kappa \to *. \lambda \text{f} : (\Pi \alpha : \kappa.\, \tau' \alpha). \lambda \beta : \kappa.\, \text{f}\, \beta$$

The first two abstractions of polyapp$_\Box$ bind the components of an arbitrary $(\Box, *)$ product type. The third abstraction binds a term of the corresponding product type. The final abstraction binds the type argument. The type of polyapp$_\Box$ is:

$$\Pi \kappa : \Box.\, \Pi \tau' : \kappa \to *.\, (\Pi \alpha : \kappa.\, \tau'\, \alpha) \to \Pi \beta : \kappa.\, \tau'\, \beta$$

Lemma 3.2. polyapp$_\Box$ *is a polymorphic application function for $(\Box, *)$ products in System U.*

Proof. We have already seen that polyapp$_\Box$ is legal and has the required form. It is easily checked that $\kappa : \Box, \tau' : \kappa \to * \vdash \kappa : \Box$. Let $(\Pi \alpha : \kappa_1 . \sigma)$ be a closed, legal $(\Box, *)$ product in System U. Then κ_1 is closed and classified by $\Box$. By Theorem 3.4, we have that $(\lambda \alpha : \kappa_1.\, \sigma)$ is closed and has kind $\kappa_1 \to *$. It is easily checked that we can derive $\langle\rangle \vdash$ polyapp$_\Box$ κ_1 $(\lambda \alpha : \kappa_1.\, \sigma) : (\Pi \alpha : \kappa_1.\, \sigma) \to (\Pi \alpha : \kappa_1.\, \sigma)$, as required. □

Kind applications. Our last polymorphic application function polyapp$_\Delta$ applies terms to kinds. Terms that can be applied to types have $(\Delta, *)$ product types of the form $\Pi \chi : \Box.\, \tau$, where χ ranges over kinds and τ is a type. An application function for terms of a particular $(\Delta, *)$ product type $\Pi \chi : \Box.\, \tau$ should have the type:

$$(\Pi \chi_1 : \Box . \tau) \to \Pi \chi_2 : \Box . \tau[\chi_1 := \chi_2] \qquad (\ddagger)$$

This type is similar to $(\dagger)$, with one important difference: whereas the type variables α and β are classified by an arbitrary kind κ, the kind variables χ_1 and χ_2 can only be classified by $\Box$ (by Theorem 3.1). We express the substitution as $(\lambda \chi_1 : \Box.\, \tau)\, \chi_2$. Letting τ' denote $\lambda \chi_1 : \Box.\, \tau$, we can write $(\ddagger)$ as:

$$(\Pi \chi_1 : \Box.\, \tau'\, \chi_1) \to \Pi \chi_2 : \Box.\, \tau'\, \chi_2$$

Theorem 3.4 states that τ' has kind $\Box \to *$. Therefore, we can implement a polymorphic application function for $(\Delta, *)$ as:

$$\text{polyapp}_\Delta = \lambda \tau' : \Box \to *.\, \lambda \text{x} : (\Pi \chi_1 : \Box.\, \tau'\, \chi_1).\, \lambda \chi_2 : \Box.\, \text{x}\, \chi_2$$

The first abstraction of polyapp$_\Delta$ is a type abstraction that binds the sole component of an arbitrary $(\Delta, *)$ product type. The second abstraction binds a term of the corresponding product type. The final abstraction binds the kind argument. The type of polyapp$_\Delta$ is:

$$\text{polyapp}_\Delta : \Pi \tau' : \Box \to *.\, (\Pi \chi_1 : \Box.\, \tau'\, \chi_1) \to \Pi \chi_2 : \Box.\, \tau'\, \chi_2$$

Lemma 3.3. polyapp$_\Delta$ *is a polymorphic application function for $(\Delta, *)$ products in System U.*

Proof. We have already seen that polyapp$_\Delta$ is legal and has the required form. It is an axiom of System U that $\Box$ is classified by Δ. Let $(\Pi \chi : A.\sigma)$ be a closed, legal $(\Delta, *)$ product in System U. Then A is closed and classified by Δ. By Theorem 3.1, $A = \Box$. By Theorem 3.4, we have that $(\lambda \chi : \Box.\, \sigma)$ is closed and has kind $\Box \to *$. It is easily checked that we can derive $\langle\rangle \vdash$ polyapp$_\Delta$ $(\lambda \chi : \Box.\, \sigma) : (\Pi \chi : \Box.\, \sigma) \to (\Pi \chi : \Box.\, \sigma)$, as required. □

It is notable that the definition of polyapp$_\Delta$ uses every rule of System U. The abstractions over x, τ', and χ_2 are formed by $(*, *)$, $(\Box, *)$, and $(\Delta, *)$, respectively. The type constructor π_Δ is type-level function formed by $(\Box, \Box)$, and the kind $\Box \to *$ of τ' is formed by $(\Delta, \Box)$.

Theorem 3.5. *System U solves the Polymorphic Application Problem.*

Proof. Lemmas 3.1, 3.2, and 3.3 show that there exist legal polymorphic application functions in System U for each of its $(s, *)$ rules: $(*, *)$, $(\Box, *)$, and $(\Delta, *)$. □

Our polyapp functions we rely on two properties of System U: that we can decompose product types (Theorem 3.3), and that $\Box$ is the only subject classified by Δ (Theorem 3.1). System U appears to be a local minimum (excluding the trivial PTS with $\mathcal{R} = \emptyset$): each of the rules $(*, *)$, $(\Box, *)$, $(\Box, \Box)$, $(\Delta, *)$, $(\Delta, \Box)$ is important for our solution.

	$(*,*)$	$(\Box,*)$	$(\Delta,*)$
System F	✓	×	
System F_ω	✓	×	
System F^+_ω	✓	✓	×
System U	✓	✓	✓

Table 1. Polymorphic application functions in our PTSs.

3.3 Polymorphic application in other systems

That $\Box$ is the only subject classified by Δ is a key to solving the Polymorphic Application problem for System U: it avoids the need to abstract over $\Box$, which is impossible in System U. However, there is more to the story – in System F, $*$ is the only subject classified by $\Box$, and yet it appears that Polymorphic Application is impossible in System F. The question of whether a PTS supports Polymorphic Application seems to require whole-system consideration. We do not know of a simple test that can answer this question for an arbitrary PTS, and leave the formulation of such a test for future work.

We conjecture that Polymorphic Application is not possible for System F, F_ω, or F^+_ω. Table 1 summarizes the polymorphic application functions that can be implemented in each system using the techniques of Section 3.2. Cells marked with ✓ indicate that a polyapp_s function for $(s,*)$ products can be defined in the system. Cells marked with × indicate that the definition of polyapp_s in that system seems to be impossible. Empty cells indicate that $(s,*)$ products are outside the system.

The first column shows that polymorphic application functions for $(*,*)$ products can be implemented in all four languages. This is because a $(*,*)$ product can be decomposed into two types of kind $*$, and each language includes terms that abstract over types of kind $*$ via the $(\Box,*)$ rule.

System F does not support a $\text{polyapp}_\Box$ function for applying terms to types. In System F, decomposing $(\Box,*)$ products (i.e. quantified types) requires higher-order types, which in turn require the rule $(\Box,\Box)$.

System F_ω can implement a polymorphic application function for System F $(\Box,*)$ products, but not for its own $(\Box,*)$ products. Since all System F types have kind $*$, all $(\Box,*)$ products in System F have the form $\Pi\alpha:*.\ \tau$. These can be decomposed into a single component $\lambda\alpha:*.\ \tau$, which has kind $*\to*$ in F_ω. Since F_ω includes higher kinds, a $\text{polyapp}_\Box$ function for F_ω should abstract over the kind, as we did in the $\text{polyapp}_\Box$ for System U. Since F_ω does not include the rule $(\Delta,*)$ needed for kind abstraction in terms, it cannot implement $\text{polyapp}_\Box$.

System F^+_ω can implement $\text{polyapp}_\Box$ because it includes the rule $(\Delta,*)$. However, it can't implement polyapp_Δ because $(\Delta,*)$ product types can't be decomposed in F^+_ω. Decomposing $(\Delta,*)$ products requires kind-polymorphic types, which in turn require the rule $(\Delta,\Box)$.

The techniques used to implement these polymorphic application functions can also be used to build typed representations. Thus, we can interpret the results of Table 1 to mean: System F can be represented in F_ω, F_ω can be represented in F^+_ω, F^+_ω can be represented in U, and U can be represented in itself. In sections 4 and 5 we will use the techniques from this section to build representations of types and terms.

4. Representing Types

In this section, we define a process for representing types of kind $*$. Type representations are themselves types. We are primarily interested in term representations, and the purpose of type representations is to enable more typed operations on term representations – in particular, operations like CPS transformation that transform the type of a program in a non-trivial way.

Since the purpose of type representations is to support typed operations on term representations, we only represent types of kind $*$, which are the types of terms. We do not represent higher-order types like $\lambda\alpha:*.\ \alpha$, which has the kind $*\to*$. We only represent types in normal form – products, variables, and applications of variables to one or more types. While the terms of System U are not strongly normalizing, the types are. This ensures that any type of kind $*$ can be normalized and represented. While we only represent closed terms, it is important that we can represent open types. This is because we will represent not only the type of the top-level term being represented, but also the type of each of its subterms. Type representations should support type functions that depend on the intensional structure of their inputs. These play an important role in the implementation of our typed CPS transformation in Section 6.3. We summarize the requirements for our representation of types below.

Definition 4.1. *The requirements for our type representation procedure are:*

- *Only legal types can be represented.*
- *Every legal normal-form type of kind $*$ can be represented.*
- *Type representations support operations that fold over the structure of the type.*

Constructors for Type Representations. We represent types using higher order abstract syntax (HOAS), inspired by [27] and [30]. Type representations are types of kind U, which is defined inductively from four constructors. Theorem 3.2 states that a type of kind $*$ is either an application of a type variable to zero or more arguments, or a product derived from one of the rules $(*,*)$, $(\Box,*)$, or $(\Delta,*)$. Our type representation includes a constructor for each case.

Definition 4.2 (Constructors of Type Representations)**.** *The kind* U *is defined inductively by the constructors:*

$$\begin{array}{ll}\langle\rangle \vdash \text{Var} & :*\to\text{U}\\ \langle\rangle \vdash \text{Prod}_* & :\text{U}\to\text{U}\to\text{U}\\ \langle\rangle \vdash \text{Prod}_\Box & :(\Pi\chi:\Box.\ (\chi\to\text{U})\to\text{U})\\ \langle\rangle \vdash \text{Prod}_\Delta & :(\Box\to\text{U})\to\text{U}\end{array}$$

The constructor Var builds representations of type variables applied to zero or more types. The constructors of Prod_*, $\text{Prod}_\Box$ and Prod_Δ build representations of $(*,*)$, $(\Box,*)$ and $(\Delta,*)$ products, respectively. Their types are similar to those of the constructors π_*, $\pi_\Box$, and π_Δ defined in Section 3.1, except that they construct types of kind U instead of types kind $*$. The body of this paper will keep the definitions of U and its constructors abstract. The appendix of the full paper gives concrete definitions of U and its constructors as System U terms.

Building Type Representations. The procedure $\triangleright$ for building type representations is defined in Figure 4. It takes as input the derivation of a normal form type of kind $*$ and outputs a representation of the type. The representation of a product type depends on whether it is a $(*,*)$ product, a $(\Box,*)$ product, or a $(\Delta,*)$ product.

An application of a variable to zero or more types is represented by applying the constructor Var to it. A product formed by $(*,*)$ has the form $\tau_1\to\tau_2$, where τ_1 and τ_2 are each of kind $*$. It is represented by applying the constructor Prod_* to the representations of τ_1 and τ_2. A product formed by $(\Box,*)$ has the form $\Pi\alpha:\kappa.\ \tau$, where τ has kind $*$ and α may occur free in τ. We build the representation of τ in the environment $\Gamma,\alpha:\kappa$, and abstract over α in the representation σ. The result has kind $\kappa\to$ U in the environment Γ. We then apply the constructor $\text{Prod}_\Box$ to the kind κ and the resulting abstraction. A product formed by $(\Delta,*)$ has the form

$$\frac{}{\Gamma \vdash \alpha\ A_1\ \ldots A_n : * \rhd \text{Var}\ (\alpha\ A_1\ \ldots\ A_n)}$$

$$\frac{\Gamma \vdash \tau_1 : * \rhd \sigma_1 \qquad \Gamma \vdash \tau_2 : * \rhd \sigma_2}{\Gamma \vdash \tau_1 \to \tau_2 : * \rhd \text{Prod}_*\ \sigma_1\ \sigma_2}$$

$$\frac{\Gamma \vdash \kappa : \Box \qquad \Gamma, \alpha : \kappa \vdash \tau : * \rhd \sigma}{\Gamma \vdash (\Pi\alpha : \kappa.\tau) : * \rhd \text{Prod}_\Box\ \kappa\ (\lambda\alpha : \kappa.\sigma)}$$

$$\frac{\Gamma, \chi : \Box \vdash \tau : * \rhd \sigma}{\Gamma \vdash (\Pi\chi : \Box.\tau) : * \rhd \text{Prod}_\Delta\ (\lambda\chi : \Box.\sigma)}$$

Figure 4. Type Representation Procedure

$\Pi\chi : \Box.\ \tau$, where τ has kind $*$ and χ may occur free in τ. We build the representation of τ in the environment $\Gamma, \chi : \Box$, and abstract over χ in the representation σ. The result has kind $\Box \to \text{U}$ in the environment Γ. We then apply the constructor Prod_Δ to the resulting abstraction.

Theorem 4.1 (Kinds of type representations)**.** *If* $\Gamma \vdash \tau : *$ *and* $\Gamma \vdash \tau : * \rhd \sigma$*, then* $\Gamma \vdash \sigma : \text{U}$.

Example 4.1. The type of the polymorphic identity function, $\Pi\alpha : *.\alpha \to \alpha$, is represented as $\text{Prod}_\Box\ *\ (\lambda\alpha : *.\ \text{Prod}_*\ (\text{Var}\ \alpha)\ (\text{Var}\ \alpha))$.

When context Γ of the derivation $\Gamma \vdash \tau : *$ is clear, we write $\overline{\tau}$ to denote the type σ such that $\Gamma \vdash \tau : * \rhd \sigma$.

Folds over type representations. Our type representation enables operations that fold over the structure of the type. A fold is defined by supplying case functions for each case of the structure of types of kind $*$: variables (or type applications with a variable in function position), $(*, *)$ products, $(\Box, *)$ products, and $(\Delta, *)$ products.

Case functions for type variables have kind $* \to *$. Variables are the base case for our inductive type representation. The fold function maps an input variable to some type of kind $*$. Since we only represent types of kind $*$, the input type variable must have kind $*$. Case functions for $(*, *)$ products of the form $\tau_1 \to \tau_2$ have kind $* \to * \to *$. Its two arguments of kind $*$ correspond the results of folding over τ_1 and τ_2. Case functions for $(\Box, *)$ products of the form $\Pi\alpha : \kappa.\ \tau$ have kind $\Pi\chi : \Box.\ (\chi \to *) \to *$. An argument type of kind $(\chi \to *)$ will be the result of folding over a type in which a type variable of kind χ can occur free. Case functions for $(\Delta, *)$ products of the form $\Pi\chi : \Box.\ \tau$ have kind $(\Box \to *) \to *$. An argument type of kind $(\Box \to *)$ will be the result of folding over a type in which a kind variable can occur free.

Definition 4.3. *Suppose* var, prod_*, $\text{prod}_\Box$, *and* prod_Δ *satisfy:*

$$\begin{array}{ll} \langle\rangle \vdash \text{var} & : * \to * \\ \langle\rangle \vdash \text{prod}_* & : * \to * \to * \\ \langle\rangle \vdash \text{prod}_\Box & : \Pi\chi : \Box.\ (\chi \to *) \to * \\ \langle\rangle \vdash \text{prod}_\Delta & : (\Box \to *) \to * \end{array}$$

Then $\text{Fold}[\text{var}, \text{prod}_*, \text{prod}_\Box, \text{prod}_\Delta]$ *denotes the type* F *such that* $\langle\rangle \vdash \text{F} : \text{U} \to *$*, and:*

$$\begin{array}{ll} \text{F}\ \overline{\tau} \equiv_\beta \text{var}\ \tau & \text{if } \tau \text{ is of the form } \alpha\ A_1\ \ldots\ A_n \\ \text{F}\ \overline{\tau} \equiv_\beta \text{prod}_*\ (\text{F}\ \overline{\tau_1})\ (\text{F}\ \overline{\tau_2}) & \text{if } \tau = \tau_1 \to \tau_2,\ \Gamma \vdash \tau_1 : * \\ \text{F}\ \overline{\tau} \equiv_\beta \text{prod}_\Box\ \kappa\ (\lambda\alpha : \kappa.\ \text{F}\ \overline{\tau_1}) & \text{if } \tau = \Pi\alpha : \kappa.\tau_1,\ \Gamma \vdash \kappa : \Box \\ \text{F}\ \overline{\tau} \equiv_\beta \text{prod}_\Delta\ (\lambda\chi : \Box.\ \text{F}\ \overline{\tau_1}) & \text{if } \tau = \Pi\chi : \Box.\ \tau_1 \end{array}$$

Our first example of an operation on type representations is listed in Figure 5. UId recovers a type from its representation. The

$$\text{UId} = \text{Fold}[(\lambda\alpha : *.\ \alpha), \pi_*, \pi_\Box, \pi_\Delta]$$

Figure 5. A function that recovers a type from its representation

case function for variables is the identity. The case function for each product type is the corresponding constructor.

Theorem 4.2. If $\Gamma \vdash \tau : *$, then $\text{UId}\ \overline{\tau} \equiv_\beta \tau$.

We define the components of type representations similarly to the components of types:

Definition 4.4 (Components of product representations)**.**

- *The components of the representation of a* $(*, *)$ *product type* $\overline{\tau_1 \to \tau_2}$ *are* $\overline{\tau_1}$ *and* $\overline{\tau_2}$.
- *The components of the representation of a* $(\Box, *)$ *product type* $\overline{\Pi\alpha : \kappa.\ \tau}$ *are* κ *and* $\lambda\alpha : \kappa.\ \overline{\tau}$.
- *The component of a* $(\Delta, *)$ *product type* $\Pi\chi : \Box.\ \tau$ *is* $\lambda\chi : \Box.\ \overline{\tau}$.

5. Representing Terms

In this section we define a process quote$(\cdot)$ that builds representations of terms. We begin by establishing the requirements for our representation. First and foremost,we should be able to represent every legal term in the language, and representations should themselves be legal terms. All representations should be strongly normalizing, even if they represent a non-normalizing term. In order to be considered useful, we require our representations to support operations that fold over the structure of the term. We summarize our requirements typed representation of terms below.

Definition 5.1 (Requirements for term representation)**.**

- *Only legal terms can be represented.*
- *Every closed legal term can be represented.*
- *All representations are strongly normalizing.*
- *Representations support folds.*

Given these requirements, what is required to type check representations? Since a representation has different semantics than the term it represents, we expect its type to also be different. On the other hand, we expect the types of a term and its representation to be related. This allows typed operations with result types that depend on the type of the input term.

5.1 Representation using PHOAS

We represent terms using typed Parametric Higher Order Abstract Syntax (PHOAS) [12, 35]. The use of PHOAS allows our representation to support multiple operations with different result types. Recall that our type representation, which uses a simpler non-parametric HOAS, only supports operations that produce results of kind $*$. In each case we choose the simplest representation for our needs.

Our representations have types of the form $\text{Exp}\ \overline{\tau}$, where $\overline{\tau}$ is a type representation. The type Exp is defined in Figure 6. It is parametric in a type R of kind $\text{U} \to *$, which is supplied by each operation and determines the result type of the operation. Instantiating a representation of type $\text{Exp}\ \overline{\tau}$ with a result type R yields the type $\text{PExp R}\ \overline{\tau}$, which can be read "Exp $\overline{\tau}$ specialized to parameter R".

The specialized representation type PExp is inductively defined by the constructors listed in Figure 6. There is a constructor for each form of the terms of System U. System U terms are either variables, λ abstractions, or applications. The abstractions and applications can be formed by the rules $(*, *)$, $(\Box, *)$, or $(\Delta, *)$.

Our quotation procedure quote$(\cdot)$ is defined in Figure 7. It relies on a pre-quotation procedure $\blacktriangleright$ defined in Figure 8. Given a term

$$\begin{aligned}
\text{mkVar} &: \Pi \text{R} : \text{U} \to *.\ \Pi\alpha : \text{U}.\ \text{R}\ \alpha \to \text{PExp R}\ \alpha \\
\text{mkAbs}_* &: \Pi \text{R} : \text{U} \to *.\ \Pi\alpha : \text{U}.\ \Pi\beta : \text{U}. \\
&\quad (\text{R}\ \alpha \to \text{PExp R}\ \beta) \to \text{PExp R}\ (\text{Prod}_*\ \alpha\ \beta) \\
\text{mkApp}_* &: \Pi \text{R} : \text{U} \to *.\ \Pi\alpha : \text{U}.\ \Pi\beta : \text{U}. \\
&\quad \text{PExp R}\ (\text{Prod}_*\ \alpha\ \beta) \to \text{PExp R}\ \alpha \to \text{PExp R}\ \beta \\
\text{mkAbs}_\Box &: \Pi \text{R} : \text{U} \to *.\ \Pi\kappa : \Box.\ \Pi\alpha : (\kappa \to \text{U}). \\
&\quad (\Pi\beta : \kappa.\ \text{PExp R}\ (\alpha\ \beta)) \to \text{PExp R}\ (\text{Prod}_\Box\ \kappa\ \alpha) \\
\text{mkApp}_\Box &: \Pi \text{R} : \text{U} \to *.\ \Pi\kappa : \Box.\ \Pi\alpha : (\kappa \to \text{U}). \\
&\quad \text{PExp R}\ (\text{Prod}_\Box\ \kappa\ \alpha) \to \Pi\beta : \kappa.\ \text{PExp R}\ (\alpha\ \beta) \\
\text{mkAbs}_\Delta &: \Pi \text{R} : \text{U} \to *.\ \Pi\alpha : \Box \to \text{U}. \\
&\quad (\Pi\chi : \Box.\ \text{PExp R}\ (\alpha\ \chi)) \to \text{PExp R}\ (\text{Prod}_\Delta\ \alpha) \\
\text{mkApp}_\Delta &: \Pi \text{R} : \text{U} \to *.\ \Pi\alpha : \Box \to \text{U}. \\
&\quad \text{PExp R}\ (\text{Prod}_\Delta\ \alpha) \to \Pi\chi : \Box.\ \text{PExp R}\ (\alpha\ \chi) \\
\\
\text{Exp} = &\ \lambda\alpha : \text{U}.\ \Pi \text{R} : \text{U} \to *.\ \text{PExp R}\ \alpha
\end{aligned}$$

Figure 6. Representation Constructors

$$\frac{\langle\rangle \vdash e : \tau \blacktriangleright q}{\text{quote}(e) = \lambda\,\text{R} : \text{U} \to *.\ q}$$

Figure 7. Quotation

of type τ, the pre-quoter produces a term of type PExp R $\overline{\tau}$. Then quote$(\cdot)$ simply abstracts over R in the result.

The pre-quoter embeds type representations within term representations. This is a key to supporting operations like CPS that transform the type of their input. As is common in HOAS representations, we use abstractions to bind the free variables of a representation. For example, if $\alpha : \kappa \vdash$ e $\blacktriangleright$ q, then α may occur free in q. We close q by abstracting over α. If q has type PExp R $\overline{\tau}$, then $\lambda\alpha : \kappa$.q has type $\Pi\alpha : \kappa$. PExp R $\overline{\tau}$. This reflects that the representation has a free variable, and enables substituting for α by application.

The first rule of pre-quotation handles variables. Representations of variables are constructed using mkVar. Variables are represented metacircularly [28], that is, using other variables. In particular, a variable of type τ is represented using a variable of type R $\overline{\tau}$.

Abstractions formed by $(*, *)$ bind term variables in terms, and have types of the form $\tau_1 \to \tau_2$. Their representations are constructed using mkAbs$_*$. The types $\overline{\tau_1}$ and $\overline{\tau_2}$ are the components of $\overline{\tau_1 \to \tau_2}$. The premise Γ, x : $\tau_1 \vdash$ e : $\tau_2 \blacktriangleright$ q builds a representation of the body in the extended environment Γ, x : τ_1. The abstraction λx : R τ_1. q binds the free variable x in the representation of e.

Applications formed by $(*, *)$ apply terms to terms. An application e_1 e_2, where e_1 has the type $\tau_2 \to \tau$, is represented by applying the constructor mkApp$_*$ to the components of $\overline{\tau_2 \to \tau}$, and the representations of e_1 and e_2.

Abstractions formed by $(\Box, *)$ bind type variables in terms, and have types of the form $\Pi\alpha : \kappa.\ \tau$. Their representations are constructed using mkAbs$_\Box$. The kind κ and type $\lambda\alpha : \kappa.\ \overline{\tau}$ are the components of $\overline{\Pi\alpha : \kappa.\ \tau}$. The premise $\Gamma, \alpha : \kappa \vdash$ e : $\tau \blacktriangleright$ q builds a representation of the body in the extended environment

$$\frac{\Gamma \vdash \tau : *}{\Gamma \vdash x : \tau \blacktriangleright \text{mkVar R}\ \overline{\tau}\ x}$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma, x : \tau_1 \vdash e : \tau_2 \blacktriangleright q}{\Gamma \vdash (\lambda x : \tau_1.e) : \tau_1 \to \tau_2 \blacktriangleright \text{mkAbs}_*\ \text{R}\ \overline{\tau_1}\ \overline{\tau_2}\ (\lambda x : R\ \overline{\tau_1}.q)}$$

$$\frac{\Gamma \vdash \tau_2 : * \qquad \Gamma \vdash e_1 : \tau_2 \to \tau \blacktriangleright q_1 \qquad \Gamma \vdash e_2 : \tau_2 \blacktriangleright q_2}{\Gamma \vdash e_1\ e_2 : \tau \blacktriangleright \text{mkApp}_*\ \text{R}\ \overline{\tau_2}\ \overline{\tau}\ q_1\ q_2}$$

$$\frac{\Gamma \vdash \kappa : \Box \qquad \Gamma, \alpha : \kappa \vdash e : \tau \blacktriangleright q}{\Gamma \vdash (\lambda\alpha : \kappa.e) : (\Pi\alpha : \kappa.\tau) \blacktriangleright \text{mkAbs}_\Box\ \text{R}\ \kappa\ (\lambda\alpha : \kappa.\overline{\tau})\ (\lambda\alpha : \kappa.q)}$$

$$\frac{\Gamma \vdash \kappa : \Box \qquad \Gamma \vdash e : (\Pi\alpha : \kappa.\tau) \blacktriangleright q \qquad (\overline{\tau}[\alpha := \tau_1]) \rightsquigarrow (\overline{\tau[\alpha := \tau_1]}) = c}{\Gamma \vdash e\ \tau_1 : (\tau[\alpha := \tau_1]) \blacktriangleright c\ (\text{mkApp}_\Box\ \text{R}\ \kappa\ (\lambda\alpha : \kappa.\overline{\tau})\ q\ \tau_1)}$$

$$\frac{\Gamma, \chi : \Box \vdash e : \tau \blacktriangleright q}{\Gamma \vdash (\lambda\chi : \Box.e) : (\Pi\chi : \Box.\tau) \blacktriangleright \text{mkAbs}_\Delta\ \text{R}\ (\lambda\chi : \Box.\overline{\tau})\ (\lambda\chi : \Box.q)}$$

$$\frac{\Gamma \vdash e : (\Pi\chi : \Box.\tau) \blacktriangleright q}{\Gamma \vdash e\ \kappa : (\tau[\chi := \kappa]) \blacktriangleright \text{mkApp}_\Delta\ \text{R}\ (\lambda\chi : \Box.\overline{\tau})\ q\ \kappa}$$

Figure 8. Pre-quotation

$\Gamma, \alpha : \kappa$. The abstraction $\lambda\alpha : \kappa$. q binds the free variable α in the representation of e.

Applications formed by $(\Box, *)$ apply terms to types. Only the term is represented; the type argument is not represented, even if it is of kind $*$. The constructor mkApp$_\Box$ is applied to the components of the $\overline{\Pi\alpha : \kappa.\ \tau}$, the representation q of the term e, and the type argument τ_1. The result is a term of type PExp R $\overline{\tau}[\alpha := \tau_1]$. A coercion c of type PExp R $(\overline{\tau}[\alpha := \tau_1]) \to$ PExp R $(\overline{\tau[\alpha := \tau_1]})$ is generated by the binary operation $\rightsquigarrow$. Coercions are discussed further below, and full detail is given in the appendix to the full paper.

Abstractions formed by $(\Delta, *)$ bind kind variables in terms, and have types of the form $\Pi\chi : \Box.\ \tau$. Their representations are constructed using mkAbs$_\Delta$. The type $\lambda\chi : \Box.\ \overline{\tau}$ is the component of $\overline{\Pi\chi : \Box.\ \tau}$. The premise $\Gamma, \chi : \Box \vdash$ e : $\tau \blacktriangleright$ q builds a representation of the body in the extended environment $\Gamma, \chi : \Box$. The abstraction $\lambda\chi : \Box$. q binds the free variable χ in the representation of e.

Applications of products formed by $(\Delta, *)$ apply terms to kinds. Again, only the term is represented. The constructor mkApp$_\Delta$ is applied to the component of $\overline{\Pi\chi : \Box.\ \tau}$, the representation of the term, and the kind argument.

Example 5.1. *Let id =* $\lambda\alpha : *.\lambda x : \alpha.x$.

$$\begin{aligned}
\text{quote(id)} = &\lambda \text{R} : \text{U} \to *. \\
&\text{mkAbs}_\Box\ \text{R}\ *\ (\lambda\alpha : *.\ \overline{\alpha \to \alpha}) \\
&\quad (\lambda\alpha : *.\ \text{mkAbs}_*\ \text{R}\ \overline{\alpha}\ \overline{\alpha} \\
&\qquad (\lambda x : \text{R}\ \overline{\alpha}.\ \text{mkVar R}\ \overline{\alpha}\ x))
\end{aligned}$$

In bottom-up order, the mkVar term corresponds to the output of the pre-quoter $\blacktriangleright$ on the derivation $\alpha : *, x : \alpha \vdash x : \alpha$, the mkAbs$_*$ term to the output on $\alpha : * \vdash (\lambda x : \alpha.\ x) : \alpha \to \alpha$, and the mkAbs$_\Box$ term to the output on $\langle\rangle \vdash$ id : $(\Pi\alpha : *.\alpha \to \alpha)$. At the top-level, quote(id) abstracts over R in the pre-quotation of id.

For convenience, we define a notation $\bar{\text{e}}$ as we did for type representations, though its definition is slightly different. When e

is a term, $\overline{\mathrm{e}}$ denotes its pre-quotation, which allows us to use $\overline{\mathrm{e}}$ even when e is not closed.

$$\frac{\Gamma \vdash \mathrm{e} : \tau \blacktriangleright \mathrm{q} \qquad \langle\rangle \vdash \mathrm{F} : \mathrm{U} \to *}{\overline{\mathrm{e}} = q[\mathrm{R} := \mathrm{F}]}$$

We allow the environment Γ and the result type function F to be implied by the context.

Since variables are represented by variables with different types, we define a representation environment $\overline{\Gamma}$ in which pre-quotations are legal. We define $\overline{\Gamma}$ inductively by the following rules. We allow the result type function F to be implied by context.

Definition 5.2 (Representation Environment)**.**

$$\begin{array}{rll} \overline{\langle\rangle} = \langle\rangle & & \\ \overline{\Gamma, \mathrm{x} : \tau} = \overline{\Gamma}, \mathrm{x} : \mathrm{F}\,\overline{\tau} & \text{if } \Gamma \vdash \tau : * \\ \overline{\Gamma, \alpha : \kappa} = \overline{\Gamma}, \alpha : \kappa & \text{if } \Gamma \vdash \kappa : \Box \\ \overline{\Gamma, \chi : \Box} = \overline{\Gamma}, \chi : \Box & \end{array}$$

We now formalize the types of pre-quotations and quotations:

Theorem 5.1. *If* $\Gamma \vdash \mathrm{e} : \tau$ *and* $\langle\rangle \vdash \mathrm{F} : \mathrm{U} \to *$*, then* $\overline{\Gamma} \vdash \overline{\mathrm{e}} : \mathrm{PExp\ F}\ \overline{\tau}$.

Theorem 5.2 (Types of quotations)**.** *If* $\langle\rangle \vdash e : \tau : *$*, and* $\mathrm{quote(e)} = \mathrm{q}$*, then* $\langle\rangle \vdash \mathrm{q} : \mathrm{Exp}\,\overline{\tau}$.

Theorem 5.3. *If* $\mathrm{quote(e)} = \mathrm{q}$*, then* q *is strongly normalizing.*

Authors traditionally define a representation in λ-calculus to be a normal form [22, 23, 27]. We follow Pfenning and Lee [26] and define constructors for our representation, which allow us to abstract away the details of our encoding. Representations built from our constructors are not normal forms, but reduce to normal forms in a few predictable steps. We provide an example in the appendix to the full paper [1]. It is also possible to define a quoter that produces closed normal forms.

5.2 Tying the knot

Theorem 5.2 states that our quotation procedure is complete: every legal System U term can be represented. We achieve self-representation using the techniques developed for our solution to the Polymorphic Application problem in Section 3.2. The type of each $\mathrm{polyapp}_s$ function is related to the types of the corresponding representation constructors mkAbs_s and mkApp_s.

Each polymorphic application function $\mathrm{polyapp}_s$ abstracts over the components of $(s, *)$ product types. We can use UId to define a version of $\mathrm{polyapp}_s$ that abstracts over the components of $(s, *)$ type representations. For example, $\mathrm{polyapp}_\Delta$ could be defined as:

$$\lambda\alpha : \Box \to \mathrm{U}.\ \lambda x : \mathrm{UId}\ (\mathrm{Prod}_\Delta\ \alpha).\ \lambda\chi : \Box.\ x\ \chi$$

The application of x to χ is legal because $\mathrm{UId}\ (\mathrm{Prod}_\Delta\ \alpha)$ is equivalent to $\Pi\chi_1 : \Box.\ \mathrm{UId}\ (\alpha\ \chi_1)$. This version of $\mathrm{polyapp}_\Delta$ can have either of the following equivalent types:

- $\Pi\alpha : \Box \to \mathrm{U}.\ (\Pi\chi : \Box.\ \mathrm{UId}\ (\alpha\ \chi)) \to \mathrm{UId}\ (\mathrm{Prod}_\Delta\ \alpha)$
- $\Pi\alpha : \Box \to \mathrm{U}.\ \mathrm{UId}\ (\mathrm{Prod}_\Delta\ \alpha) \to \Pi\chi : \Box.\ \mathrm{UId}\ (\alpha\ \chi)$

If we replace UId with PExp R in the first type and abstract over R, we get the type of mkAbs_Δ. The same operation on the second type yields the type of mkApp_Δ.

We summarize the results of Section 3.2, Section 4, and Section 5 as follows: Every form of product type in System U can be decomposed, and we can implement a polymorphic application function by abstracting over the components. Further, every form of product type in System U can be represented. Type representations can also be decomposed and the components can be also used to define a polymorphic application function. Finally, we can combine our polymorphic application functions with standard representation techniques to achieve self-representation.

$$\begin{array}{rl} \mathrm{Abs}_* = & \lambda\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}. \\ & (\mathrm{R}\ \alpha \to \mathrm{R}\ \beta) \to \mathrm{R}\ (\mathrm{Prod}_*\ \alpha\ \beta) \\ \mathrm{App}_* = & \lambda\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}. \\ & \mathrm{R}\ (\mathrm{Prod}_*\ \alpha\ \beta) \to \mathrm{R}\ \alpha \to \mathrm{R}\ \beta \end{array}$$

$$\begin{array}{rl} \mathrm{Abs}_\Box = & \lambda\mathrm{R} : \mathrm{U} \to *.\ \Pi\kappa : \Box.\ \Pi\alpha : (\kappa \to \mathrm{U}). \\ & (\Pi\beta : \kappa.\ \mathrm{R}\ (\alpha\ \beta)) \to \mathrm{R}\ (\mathrm{Prod}_\Box\ \kappa\ \alpha) \\ \mathrm{App}_\Box = & \lambda\mathrm{R} : \mathrm{U} \to *.\ \Pi\kappa : \Box.\ \Pi\alpha : (\kappa \to \mathrm{U}). \\ & \mathrm{R}\ (\mathrm{Prod}_\Box\ \kappa\ \alpha) \to \Pi\beta : \kappa.\ \mathrm{R}\ (\alpha\ \beta) \end{array}$$

$$\begin{array}{rl} \mathrm{Abs}_\Delta = & \lambda\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \Box \to \mathrm{U}. \\ & (\Pi\chi : \Box.\ \mathrm{R}\ (\alpha\ \chi)) \to \mathrm{R}\ (\mathrm{Prod}_\Delta\ \alpha) \\ \mathrm{App}_\Delta = & \lambda\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \Box \to \mathrm{U}. \\ & \mathrm{R}\ (\mathrm{Prod}_\Delta\ \alpha) \to \Pi\chi : \Box.\ \mathrm{R}\ (\alpha\ \chi) \end{array}$$

Figure 9. Types of Case Functions

5.3 Folds over term representations

Our representation of terms is designed to support operations that fold over the structure of the term. A fold is defined by six case functions, one each for abstractions and applications formed by $(*, *)$, $(\Box, *)$, and $(\Delta, *)$. The result of a fold is defined by induction on the structure of the term. For each term, the corresponding case function is applied to the the results of folding over its subterms. This is made formal below.

The types of the case functions of a fold are defined in 9. The types App_*, $\mathrm{App}_\Box$, and App_Δ and are similar to the types of our Polymorphic Application functions $\mathrm{polyapp}_*$, $\mathrm{polyapp}_\Box$ and $\mathrm{polyapp}_\Delta$ from Section 3.2. Each App_s types and the type of each $\mathrm{polyapp}_s$ function relies on the idea of decomposition. The difference is that the App_s types deal with components of type representations, while the types of the $\mathrm{polyapp}_s$ functions deal with components of product types.

The specification of an operation on term representations consists of a result type R, a witness of type Witness R, and six case functions. The witness ensures that for all types τ and τ_1 such that $\Gamma, \alpha : \kappa \vdash \tau : *$ and $\Gamma \vdash \tau_1 : \kappa$, the quoter can synthesize a coercion of type: $\mathrm{PExp\ R}\ (\overline{\tau}[\alpha := \tau_1]) \to \mathrm{PExp\ R}\ (\overline{\tau[\alpha := \tau_1]})$. These coercions are necessary in order to represent type applications. The semantics of a coercion thus depends on the witness, which gives us the flexibility needed to support multiple operations on a single generic representation. We will say more about the coercions for each of our operations in the following section. Witnesses and coercions are described in greater detail in the appendix to the full paper.

Definition 5.3. Suppose F, w, abs_*, app_*, $\mathrm{abs}_\Box$, $\mathrm{app}_\Box$, abs_Δ, and app_Δ satisfy:

$$\begin{array}{lll} \langle\rangle \vdash \mathrm{F} & : \mathrm{U} \to * & \\ \langle\rangle \vdash \mathrm{w} & : \mathrm{Witness\ F} & \\ \langle\rangle \vdash \mathrm{abs}_* & : \mathrm{Abs}_*\ \mathrm{F} & \langle\rangle \vdash \mathrm{app}_* : \mathrm{App}_*\ \mathrm{F} \\ \langle\rangle \vdash \mathrm{abs}_\Box & : \mathrm{Abs}_\Box\ \mathrm{F} & \langle\rangle \vdash \mathrm{app}_\Box : \mathrm{App}_\Box\ \mathrm{F} \\ \langle\rangle \vdash \mathrm{abs}_\Delta & : \mathrm{Abs}_\Delta\ \mathrm{F} & \langle\rangle \vdash \mathrm{app}_\Delta : \mathrm{App}_\Delta\ \mathrm{F} \end{array}$$

Then $\mathrm{fold[F, w, abs}_*, \mathrm{app}_*, \mathrm{abs}_\Box, \mathrm{app}_\Box, \mathrm{abs}_\Delta, \mathrm{app}_\Delta]$ denotes a term f such that $\langle\rangle \vdash \mathrm{f} : \Pi\alpha : \mathrm{U.PExp\ F}\ \alpha \to \mathrm{F}\ \alpha$, and for any context Γ, term e, and type τ such that $\Gamma \vdash \mathrm{e} : \tau$, we have that:

$$
\begin{array}{ll}
\mathrm{w} & : \text{Witness UId} \\
\mathrm{id}_* & = \lambda\alpha : \mathrm{U}.\ \lambda\beta : \mathrm{U}.\ \lambda \mathrm{x} : \mathrm{UId}\ \alpha \to \mathrm{UId}\ \beta.\ \mathrm{x} \\
\mathrm{id}_\Box & = \lambda\kappa : \Box.\ \lambda\alpha : \kappa \to \mathrm{U}.\ \lambda \mathrm{x} : (\Pi\beta : \kappa.\ \mathrm{UId}\ (\alpha\ \beta)).\ \mathrm{x} \\
\mathrm{id}_\Delta & = \lambda\alpha : \Box \to \mathrm{U}.\ \lambda \mathrm{x} : (\Pi\chi : \Box.\ \mathrm{UId}\ (\alpha\ \chi)).\ \mathrm{x} \\
\mathrm{unquote} & = \mathrm{fold}[\mathrm{UId}, \mathrm{w}, \mathrm{id}_*, \mathrm{id}_*, \mathrm{id}_\Box, \mathrm{id}_\Box, \mathrm{id}_\Delta, \mathrm{id}_\Delta]
\end{array}
$$

Figure 10. Definition of unquote

If e is a variable, then $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{e}$.
If $\tau = \tau_1 \to \tau_2,\ \Gamma \vdash \tau_1 : *,$ and $\mathrm{e} = \lambda \mathrm{x} : \tau_1.\ \mathrm{e}_1$, then
$\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{abs}_*\ \overline{\tau_1}\ \overline{\tau_2}\ (\lambda \mathrm{x} : \mathrm{F}\ \overline{\tau_1}.\ \mathrm{f}\ \overline{\tau_2}\ \overline{\mathrm{e}_1})$.
If $\mathrm{e} = \mathrm{e}_1\ \mathrm{e}_2,\ \Gamma \vdash \mathrm{e}_2 : \tau_1 : *,$ then
$\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{app}_*\ \overline{\tau_1}\ \overline{\tau}\ (\mathrm{f}\ \overline{\tau_1 \to \tau}\ \overline{\mathrm{e}_1})\ (\mathrm{f}\ \overline{\tau_1}\ \overline{\mathrm{e}_2})$.
If $\tau = \Pi\alpha : \kappa.\ \tau_1,\ \Gamma \vdash \kappa : \Box,$ and $\mathrm{e} = \lambda\alpha : \kappa.\ \mathrm{e}_1$, then
$\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{abs}_\Box\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau_1})\ (\lambda\alpha : \kappa.\ \mathrm{f}\ \overline{\tau_1}\ \overline{\mathrm{e}_1})$.
If $\mathrm{e} = \mathrm{e}_1\ \tau_2,\ \Gamma \vdash \kappa : \Box,$ and $\Gamma \vdash \mathrm{e}_1 : \Pi\alpha : \kappa.\ \tau_1$, then
$\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta c\ (\mathrm{app}_\Box\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau_1})\ (\mathrm{f}\ \overline{\Pi\alpha : \kappa.\ \tau_1}\ \overline{\mathrm{e}_1})\ \tau_2)$
for some coercion c.
If $\tau = \Pi\chi : \Box.\ \tau_1,$ and $\mathrm{e} = \lambda\chi : \Box.\ \mathrm{e}_1$, then
$\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{abs}_\Delta\ (\lambda\chi : \Box.\overline{\tau_1})\ (\lambda\chi : \Box.\ \mathrm{f}\ \overline{\tau_1}\ \overline{\mathrm{e}_1})$.
If $\mathrm{e} = \mathrm{e}_1\ \kappa,\ \Gamma \vdash \kappa : \Box,$ and $\Gamma \vdash \mathrm{e}_1 : \Pi\chi : \Box.\ \tau_1$, then
$\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{app}_\Delta\ (\lambda\chi : \Box.\ \overline{\tau_1})\ (\mathrm{f}\ \overline{\Pi\chi : \Box.\tau_1}\ \overline{\mathrm{e}_1})\ \kappa$.

Definition 5.3 states that the operation specified by fold[F, w, abs_*, app_*, $\mathrm{abs}_\Box$, $\mathrm{app}_\Box$, abs_Δ, app_Δ] has the semantics expected of a fold. The seven cases are mutually exclusive and exhaustive: a term in System U is either a variable, an abstraction, or an application. Abstractions and applications can be formed by one of three rules: $(*, *)$, $(\Box, *)$, $(\Delta, *)$.

6. Operations

In this section we show how to program three benchmark operations on our representation. The first, called unquote, is a typed self-recognizer – a self-interpreter that recovers an term from its representation. The second, called isAbs, is a simple example of an intensional predicate. It tests whether its input represents an abstraction or an application. The third, and most complex, is a typed self-applicable continuation-passing-style (CPS) transformation.

6.1 Unquote

Our self-recognizer unquote is defined in Figure 10. It produces results with types determined by UId. Each case function in the definition of unquote is an identity function. If a term e has type τ, then unquoting a representation of e produces a term of type UId $\overline{\tau}$. Theorem 4.2 states that UId $\overline{\tau}$ is equivalent to τ.

Theorem 6.1 (Type of unquote)**.**

$$\langle\rangle \vdash \mathrm{unquote} : (\Pi\alpha : \mathrm{U}.\ \mathrm{Exp}\ \alpha \to \mathrm{UId}\ \alpha)$$

Unquote folds identity functions over the term. The result is equivalent to the original term.

Theorem 6.2 (Correctness of unquote)**.**
If $\langle\rangle \vdash \mathrm{e} : \tau$ *and* $\mathrm{quote}(\mathrm{e}) = \mathrm{q}$*, then* $\mathrm{unquote}\ \overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{e}$.

The coercions produced by the witness for unquote are always identity functions. Consider the type UId $(\overline{\tau}[\alpha := \tau_1]) \to$ UId $\overline{\tau[\alpha := \tau_1]}$, which is the type of an arbitrary coercion for unquote. This type is equivalent to $\tau[\alpha := \tau_1] \to \tau[\alpha := \tau_1]$.

6.2 isAbs

Our second benchmark operation isAbs is shown in Figure 11. isAbs tests if its input is a representation of an abstraction. This demonstrates that we can define operations on a representation that

$$
\begin{array}{ll}
\mathrm{Bool} = & \Pi\alpha : *.\ \alpha \to \alpha \to \alpha \\
\mathrm{true} = & \lambda\alpha : *.\ \lambda t : \alpha.\ \lambda f : \alpha.\ t \\
\mathrm{false} = & \lambda\alpha : *.\ \lambda t : \alpha.\ \lambda f : \alpha.\ f \\
 & \\
\mathrm{UBool} & = \lambda\alpha : \mathrm{U}.\ \mathrm{Bool} \\
 & \\
\mathrm{w} & : \text{Witness UBool} \\
\mathrm{abs}_* & = \lambda \mathrm{T}_1 : *.\ \lambda \mathrm{T}_2 : *.\ \lambda \mathrm{f} : \mathrm{Bool} \to \mathrm{Bool}.\mathrm{true} \\
\mathrm{app}_* & = \lambda \mathrm{T}_1 : *.\ \lambda \mathrm{T}_2 : *.\ \lambda \mathrm{f} : \mathrm{Bool}.\ \lambda \mathrm{e}_1 : \mathrm{Bool}.\ \mathrm{false} \\
\mathrm{abs}_\Box & = \lambda\ \chi : \Box.\ \lambda \mathrm{F} : \chi \to *.\ \lambda \mathrm{e}_1 : \chi \to \mathrm{Bool}.\ \mathrm{true} \\
\mathrm{app}_\Box & = \lambda\ \chi : \Box.\ \lambda \mathrm{F} : \chi \to *.\ \lambda \mathrm{e}_1 : \mathrm{Bool}.\ \lambda \mathrm{x} : \chi.\ \mathrm{false} \\
\mathrm{abs}_\Delta & = \lambda \mathrm{T}_1 : \Box \to *.\ \lambda \mathrm{e}_1 : \Box \to \mathrm{Bool}.\ \mathrm{true} \\
\mathrm{app}_\Delta & = \lambda \mathrm{T}_1 : \Box \to *.\ \lambda \mathrm{e}_1 : \mathrm{Bool}.\ \lambda\ \chi : \Box.\mathrm{false} \\
\mathrm{isAbs} & = \mathrm{fold}[\mathrm{UBool}, \mathrm{w}, \mathrm{abs}_*, \mathrm{app}_*, \mathrm{abs}_\Box, \mathrm{app}_\Box, \mathrm{abs}_\Delta, \mathrm{app}_\Delta]
\end{array}
$$

Figure 11. Specification of isAbs

cannot be defined directly on the represented term. The result type UBool of isAbs is the constant Bool function. Each case function in the definition of isAbs is a constant function. It discards the result of folding over its subterm(s), since we are only interested in the outermost constructor of the representation. The case functions for abstractions are constant true functions and the case functions for applications are constant false functions.

Theorem 6.3 (Type of isAbs)**.**

$$\langle\rangle \vdash \mathrm{isAbs} : (\Pi\alpha : \mathrm{U}.\ \mathrm{Exp}\ \alpha \to \mathrm{Bool})$$

The application of isAbs to the representation of a term of type τ produces a term of type UBool $\overline{\tau}$, which is equivalent to Bool. Like those for UId, coercions for UBool types are identity functions. Consider the type UBool $(\overline{\tau}[\alpha := \tau_1]) \to$ UBool $\overline{\tau[\alpha := \tau_1]}$, which is the type of an arbitrary coercion for isAbs. Since UBool is a constant function, this type is equivalent to Bool $\to$ Bool.

Theorem 6.4 (Correctness of isAbs)**.**
If $\langle\rangle \vdash \mathrm{e} : \tau : *$ *and* $\mathrm{quote}(\mathrm{e}) = \mathrm{q}$ *then:*

- *If* $\mathrm{e} = \lambda x : \mathrm{A}.\mathrm{e}_1$*, then* $\mathrm{isAbs}\ \overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{true}$.
- *If* $\mathrm{e} = \mathrm{e}_1\ \mathrm{A}$*, then* $\mathrm{isAbs}\ \overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{false}$.

6.3 Continuation-Passing Style

In this section, we implement a type call-by-name continuation-passing style (CPS) transformation on our representation. CPS transformation is commonly used in compilers for functional languages. It makes the evaluation order (call-by-name in our case) explicit, and eliminates the need for a control-stack. There are examples of typed CPS transformations in the literature, though ours is the first that is self-applicable. We extend the typed CPS transformation of [27], which operates on typed representations of simply typed λ calculus. To transform abstractions and applications of types and kinds, we extend the technique used by Morrisett et al [24] to transform System F type abstractions and applications.

The result of applying the CPS transformation to the representation of a term of type τ is a term of type CPS $\overline{\tau}$. The type CPS is shown in Figure 12. CPS is defined via a fold CPS_1 and a helper function Ct. The CPS-transformation itself is defined in Figure 13.

Theorem 6.5 (Type of cps)**.**

$$\langle\rangle \vdash \mathit{cps} : (\Pi\alpha : \mathit{U}.\ \mathit{Exp}\ \alpha \to \mathit{CPS}\ \alpha)$$

Coercions for CPS types are not identity functions, unlike those for UId and UBool types. As a simple example, note that for type variables α and β, CPS $(\overline{\alpha}[\alpha := \beta \to \beta])$ is not equivalent to CPS $(\overline{\alpha[\alpha := \beta \to \beta]})$. The former simplifies to CPS (Var $(\beta \to$

$$
\begin{array}{ll}
\mathrm{Ct} & = \lambda \mathrm{T} : *.\ \Pi\ \mathrm{V} : *.\ (\mathrm{T} \to \mathrm{V}) \to \mathrm{V} \\
 & \\
\mathrm{var} & = \lambda \alpha : *.\ \alpha \\
\mathrm{prod}_* & = \lambda \alpha : *.\ \lambda \beta : *.\ \mathrm{Ct}\ \alpha \to \mathrm{Ct}\ \beta \\
\mathrm{prod}_\Box & = \lambda \chi : \Box.\ \lambda \alpha : \chi \to *.\ \Pi \beta : \chi.\ \mathrm{Ct}\ (\alpha\ \beta) \\
\mathrm{prod}_\Delta & = \lambda \mathrm{T}_1 : \Box \to *.\ \Pi \chi : \Box.\ \mathrm{Ct}\ (\mathrm{T}_1\ \chi) \\
\mathrm{CPS}_1 & = \mathrm{Fold}[\mathrm{var}, \mathrm{prod}_*, \mathrm{prod}_\Box, \mathrm{prod}_\Delta] \\
 & \\
\mathrm{CPS} & = \lambda \mathrm{T} : \mathrm{U}.\ \mathrm{Ct}\ (\mathrm{CPS}_1\ \mathrm{T})
\end{array}
$$

Figure 12. The result type of CPS transformation

$$
\begin{array}{rl}
\mathrm{w} & : \mathrm{Witness\ CPS} \\
\mathrm{abs}_* & = \lambda \alpha : \mathrm{U}.\ \lambda \beta : \mathrm{U}.\ \lambda \mathrm{f} : \mathrm{CPS}\ \alpha \to \mathrm{CPS}\ \beta. \\
 & \quad \lambda \mathrm{V} : *.\ \lambda \mathrm{k} : (\mathrm{CPS}\ \alpha \to \mathrm{CPS}\ \beta) \to \mathrm{V}.\ \mathrm{k\ f} \\
 & \\
\mathrm{app}_* & = \lambda \alpha : \mathrm{U}.\ \lambda \beta : \mathrm{U}.\ \lambda \mathrm{f} : \mathrm{CPS}\ (\mathrm{Prod}_*\ \alpha\ \beta).\ \lambda \mathrm{x} : \mathrm{CPS}\ \alpha. \\
 & \quad \lambda \mathrm{V} : *.\ \lambda \mathrm{k} : (\mathrm{CPS}_1\ \beta) \to \mathrm{V}. \\
 & \quad \mathrm{f\ V}\ (\lambda \mathrm{g} : \mathrm{CPS}\ \alpha \to \mathrm{CPS}\ \beta.\ \mathrm{g\ x\ V\ k}) \\
 & \\
\mathrm{abs}_\Box & = \lambda \chi : \Box.\ \lambda \alpha : \chi \to \mathrm{U}.\ \lambda \mathrm{e} : (\Pi \beta : \chi.\ \mathrm{CPS}\ (\alpha\ \beta)). \\
 & \quad \lambda \mathrm{V} : *.\ \lambda \mathrm{k} : (\mathrm{CPS}_1\ (\mathrm{Prod}_\Box\ \chi\ \alpha)) \to \mathrm{V}.\ \mathrm{k\ e} \\
 & \\
\mathrm{app}_\Box & = \lambda \chi : \Box.\ \lambda \alpha : (\chi \to \mathrm{U}).\ \lambda \mathrm{e} : \mathrm{CPS}\ (\mathrm{Prod}_\Box\ \chi\ \alpha).\ \lambda \beta : \chi. \\
 & \quad \lambda \mathrm{V} : *.\ \lambda \mathrm{k} : (\mathrm{CPS}_1\ (\alpha\ \beta)) \to \mathrm{V}. \\
 & \quad \mathrm{e\ V}\ (\lambda \mathrm{e}_1 : (\Pi \beta_1 : \chi.\ \mathrm{CPS}\ (\alpha\ \beta_1)).\mathrm{e}_1\ \beta\ \mathrm{V\ k}) \\
 & \\
\mathrm{abs}_\Delta & = \lambda \alpha : \Box \to \mathrm{U}.\ \lambda \mathrm{e} : \Pi \chi : \Box.\ \mathrm{CPS}\ (\alpha\ \chi). \\
 & \quad \lambda \mathrm{V} : *.\ \lambda \mathrm{k} : \mathrm{CPS}_1\ (\mathrm{Prod}_\Delta\ \alpha) \to \mathrm{V}.\ \mathrm{k\ e} \\
 & \\
\mathrm{app}_\Delta & = \lambda \alpha : \Box \to \mathrm{U}.\ \lambda \mathrm{e} : \mathrm{CPS}\ (\mathrm{Prod}_\Delta\ \alpha).\ \lambda \chi : \Box. \\
 & \quad \mathrm{e\ V}\ (\lambda \mathrm{e}_1 : (\Pi \chi_1 : \Box.\ \ \mathrm{CPS}\ (\alpha\ \chi_1)).\mathrm{e}_1\ \chi\ \mathrm{V\ k}) \\
 & \\
\mathrm{cps} & = \mathrm{fold}[\mathrm{CPS}, \mathrm{w}, \mathrm{abs}_*, \mathrm{app}_*, \mathrm{abs}_\Box, \mathrm{app}_\Box, \mathrm{abs}_\Delta, \mathrm{app}_\Delta]
\end{array}
$$

Figure 13. Specification of cps

$\beta)$), and the latter to CPS (Prod_* (Var β) (Var β)). Coercions for CPS types add and remove continuations as necessary.

We don't attempt to formally verify the correctness of cps, though we validate it by testing it on the polyapp functions from Section 3.2.

7. Experiments

We conduct experiments using an implementation of System U, which is available from our website [1]. We implement a parser in Ohm, a domain specific language for writing parsers and the successor to OMeta [34]. The parser generates abstract syntax for our Haskell implementation of System U, which includes type and term quoters, a validity checker, an evaluator, and an incomplete β, η-equivalence checker. We have used the implementation to mechanically check that all System U terms, types and kinds presented in the paper are legal, and to verify the equivalence theorems. We have verified that self-applications of unquote, isAbs, and cps are legal and have normal forms. Furthermore, self-application of unquote is equivalent to unquote itself, and self-application of isAbs evaluates to the Church boolean true:

$$\frac{\text{quote(unquote) = q}}{\text{unquote}\ \overline{(\Pi\alpha : \mathrm{U}.\ \mathrm{Exp}\ \alpha \to \mathrm{UId}\ \alpha)}\ \ \mathrm{q} \equiv_{\beta,\eta} \text{unquote}}$$

$$
\mathrm{F}^*_\omega \quad \boxed{\begin{array}{ll} \mathcal{S} & *, \Box \\ \mathcal{A} & * : \Box, \Box : \Box \\ \mathcal{R} & (*, *), (\Box, *), (\Box, \Box) \end{array}}
$$

Figure 14. PTS specification of F^*_ω

$$\frac{\text{quote(isAbs) = q}}{\text{isAbs}\ \overline{(\Pi\alpha : \mathrm{U}.\ \mathrm{Exp}\ \alpha \to \mathrm{UBool}\ \alpha)}\ \ \mathrm{q} \equiv_{\beta,\eta} \mathrm{true}}$$

We have validated cps by applying it to each of our polyapp functions from Section 3.2.

8. Related Work

The problem of typed self-representation has been studied since 1991, when Pfenning and Lee considered whether System F ($\lambda 2$) could represent itself [26]. They found that "metacircularity seems to be impossible" for System F. However, they developed several typed representations of one language in another – System F in System F_ω , and F_ω in F^+_ω. Their representation technique inspired our own. They use higher order abstract syntax similar to ours, with two important differences. They don't represent types, and their quoter does not change the types of variables. Each of these is important for our typed cps transformation.

The key idea of decomposition of product types is already present in Pfenning and Lee [26]. The idea recurs throughout the literature on typed HOAS representations. In the setting of pure type systems, the pattern becomes more clear. We identify decomposition of product types and abstraction of the resulting components as key requirements for typed representation of a pure type system.

Rendel, Ostermann, and Hofer [27] defined the first typed self-representation and self-recognizer (which they called eval). They study a language F^*_ω defined in Figure 14. Like F_ω, System F^*_ω contains the rule $(\Box, \Box)$ which allows formation of higher-order types. Unlike F_ω, which classifies types using a family of kinds induced by the sort Δ and axiom $\Box : \Delta$, System F^*_ω adds an axiom $\Box : \Box$, which forms types that classify other types. Types that classify other types play the role of "kind" in System F^*_ω. This is sufficient to tie the knot: abstractions formed by $(\Box, *)$ can abstract over both types and "kinds" in terms. Similarly, $(\Box, \Box)$ can abstract over both types and "kinds" in types.

Our type representation is partly inspired by that of [27], which represents the types of simply typed λ-calculus in F^*_ω. They use type representations in a representation of simply-typed λ-calculus in F^*_ω, which supports a typed CPS transformation. Our self-representation of System U and CPS transformation are also inspired by their representation and CPS transformation of simply-typed λ-calculus.

Like System U, System F^*_ω is not normalizing. Unlike System U, type checking of F^*_ω is undecidable due to the $(\Box, \Box)$ rule. We conjecture that System U can be embedded into F^*_ω, but that System F^*_ω cannot be embedded into System U.

Our representation of types is also inspired by Saha et al. [30]. They study intensional type analysis for λ^ω_i which, like System U, includes type and kind polymorphism. They encode base types of kind Ω (analogous to $*$ in System U) using HOAS. The kinds of their HOAS type constructors parallel the kinds of the constructors of our type representations.

λ^ω_i	Kind	System U	Kind
$\to$	$\Omega \to \Omega \to \Omega$	Prod_*	$\mathrm{U} \to \mathrm{U} \to \mathrm{U}$
$\forall$	$\forall \chi.(\chi \to \Omega) \to \Omega$	$\mathrm{Prod}_\Box$	$\Pi \chi : \Box.(\chi \to \mathrm{U}) \to \mathrm{U}$
$\forall^+$	$(\forall \chi.\Omega) \to \Omega$	Prod_Δ	$(\Box \to \mathrm{U}) \to \mathrm{U}$

Despite notational differences, there is a direct correspondence between the kinds of the constructors for λ_i^ω types and our type representations. The binders $\forall$ and Π play the same role in each calculus. Furthermore, in System U $\square \rightarrow$ U is shorthand for $\Pi\chi : \square$.U (since χ does not occur free in U). A subtle difference is that there is no classifier for the kinds of λ_i^ω (they use a well-formedness condition), while in our PTS formulation of System U all kinds are classified by $\square$. Saha et al. include a type operator Typerec for intensional type analysis of base types based on folds. They support fold operations that produce higher-kinded results. The Typerec operator is primitive, which avoids the need for type representations. They did not study self-representation of System λ_i^ω, and it is an open question whether it would be possible. We conjecture that System U can be embedded into λ_i^ω, but λ_i^ω cannot be embedded into System U.

Typed representation has been extensively studied, and is still an active area of research. Chen and Xi [10, 11] studied typed representation and typed meta-programming. Carette, Kiselyov and Shan [9] use typed representation to build tagless interpretations. McBride [20] achieved a metacircular representation of dependently typed languages in Agda. Axelsson [3] developed a technique for building generic, composable typed representations as a solution to the expression problem [33]. Each of these is important related work, and we have learned from and been inspired by them, even though they did not study self-representation.

9. Future Work

Size. Our representations do not support operations that measure the size of a term. This is a limitation of our higher order abstract syntax representation. Abstractions in the representation, particularly type abstractions, can block access to the size of subterms. Assuming the size operation should produce results of some closed type Nat, we would need a way to convert a term of type $\Pi\alpha : \kappa$. Nat to Nat. The quantification α is redundant, since α does not occur free in Nat. In order to recover the Nat, we would have to apply the term of type $\Pi\alpha : \kappa$.Nat to a type of kind κ. This is not always possible, since not all System U kinds are inhabited. In [27] this was addressed by adding a type constant $\bot : \Pi\alpha : \square.\alpha$, which could be used to apply these abstractions.

*Beyond kind *.* Our representation of types is limited to types of kind *. Full representation of types is desirable, as it may eliminate the need for coercions and the witnesses that enable coercions. Full representation may also enable more operations. It is an open question whether full type representation is possible in System U.

Without type representation. At the other end of the type representation spectrum, we can consider representation of terms that don't require representation of types. We represent types in order to support our typed CPS transformation. In particular, type representations allow us to give cps the polymorphic type $\Pi\alpha$: U.Exp $\alpha \rightarrow$ CPS α. The input and output types Exp α and CPS α are both defined in terms of the quantified variable α. Our other operations, unquote and isAbs, do not require any representation of types. It is possible to define a simpler representation type Exp_1 of kind $* \rightarrow *$, and a quotation procedure that represents terms of type τ with terms of types $\text{Exp}_1\ \tau$. The representation type and quoter would be similar to those from Rendel, Ostermann, Hofer [27]. In particular, we would no longer require coercions.

Strong normalization. Of the two λ-calculi System F_ω^* and System U known to support typed self-representations, neither is strongly normalizing. It is an open question whether a language with decidable type checking and strong normalization can support typed self-representation.

Representing open terms. Our quoter changes the types of variables, which is important for our cps transformation, but also limits our representations to closed terms. Pfenning and Lee did not change the type of variables, which enabled them to represent free variables (i.e. those bound outside the representation) in the same way as bound variables. It is possible to extend our representation type with a new constructor for representing free variables, with the type:

$$\Pi\text{R} : \text{U} \rightarrow *.\Pi\alpha : *.\alpha \rightarrow \text{PExp R (Var } \alpha)$$

To represent a free variable of type τ, the quoter would first apply this constructor, producing a term of type PExp R (Var τ). Then it would synthesize a coercion to change the type to PExp R $\overline{\tau}$. Note that Var $\tau \equiv \overline{\alpha}[\alpha := \tau]$ and that $\overline{\tau} \equiv \overline{\alpha[\alpha := \tau]}$.

Dependent Types. We can extend System U with dependent types by adding to $\mathcal{R}$ the rule $(*, \square)$ that forms types that abstract over terms. The resulting system still supports polymorphic application, which indicates that it might also support self-representation. Compared to the polyapp functions for System U, the only change required is to polyapp_*, since $(*, *)$ products can now be dependent. For example, in a product type $\Pi x : \tau_1.\ \tau_2$ formed by $(*, *)$, the bound variable x can now occur free in τ_2. The polyapp_* function for the extended system could be defined as:

$$\lambda\tau_1 : *.\ \lambda\tau_2 : \tau_1 \rightarrow *.\ (\Pi x : \tau_1.\tau_2\ x) \rightarrow \Pi y : \tau_1.\ \tau_2\ y$$

Note that the kind $\tau_1 \rightarrow *$ is formed by $(*, \square)$. We still have the property that $\square$ is the only element of Δ, which is key to tying the knot. However, the addition of dependent types would raise two challenges for type representation. The first is due to the introduction of non-normalizing types (e.g. because types can contain non-normalizing terms). Our type representation in this paper only applies to types in normal form. Second, the type representation must consider how to represent types that depend on terms.

10. Conclusion

The question of whether a meaningful notion of typed self-representation is possible for a language with decidable type checking has been open since 1991 [26]. We answer in the affirmative by presenting the first typed self-representation for a λ-calculus with decidable type checking. Our calculus is System U, which was introduced in Girard's PhD thesis [16]. We embed representations of types into representations of terms, which enable operations like CPS transformation that change the type of a term. Our representation supports operations that iterate over the term, and we provide three example self-applicable operations: a typed self-recognizer that recovers a term from its representation, a predicate that tests the intensional structure of a term, and a typed CPS transformation. Ours is the first typed self-applicable CPS transformation. We have validated our results by conducting experiments using an implementation of System U in Haskell.

Acknowledgments. We thank John Bender, Mahdi Eslamimehr, Barry Jay, Todd Millstein, Cody Roux, Aaron Stump, 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/popl15/. The full paper with the appendix is available there, as are the source code for our implementation of System U and our operations.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[3] Emil Axelsson. A generic abstract syntax model for embedded languages. *SIGPLAN Not.*, 47(9):323–334, September 2012.

[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] Gilles Barthe. Type-checking injective pure type systems. *J. Funct. Program.*, 9(6):675–698, November 1999.

[7] Michel Bel. A recursion theoretic self interpreter for the lambda-calculus. http://www.belxs.com/michel/#selfint.

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

[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] 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.

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

[12] 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.

[13] Brendan Eich. Narcissus. `http://mxr.mozilla.org/mozilla/source/js/narcissus/jsexec.js`, 2010.

[14] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *J. Funct. Program.*, 1(2):155–189, 1991.

[15] Jan Herman Geuvers. *Logics and type systems*. 1993.

[16] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[17] 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.

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

[19] Oleg Mazonka and Daniel B. Cristofani. A very short self-interpreter. http://arxiv.org/html/cs/0311032v1, November 2003.

[20] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM.

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

[22] 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.

[23] Torben Æ. Mogensen. Linear-time self-interpretation of the pure lambda calculus. *Higher-Order and Symbolic Computation*, 13(3):217–237, 2000.

[24] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[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] 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.

[28] 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).

[29] Andreas Rossberg. HaMLet. http://www.mpi-sws.org/ rossberg/hamlet, 2010.

[30] Bratin Saha, Valery Trifonov, and Zhong Shao. Intensional analysis of quantified types. *ACM Trans. Program. Lang. Syst.*, 25(2):159–209, 2003.

[31] Fangmin Song, Yongsen Xu, and Yuechen Qian. The self-reduction in lambda calculus. *Theoretical Computer Science*, 235(1):171–181, March 2000.

[32] John Tromp. Binary lambda calculus and combinatory logic. In *Kolmogorov Complexity and Applications*, 2006. A Revised Version is available at http://homepages.cwi.nl/ tromp/cl/LC.pdf.

[33] Philip Wadler. The expression problem. http://www.daimi.au.dk/ madst/tool/papers/expression.txt.

[34] Alessandro Warth and Ian Piumarta. Ometa: An object-oriented language for pattern matching. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM.

[35] 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.

[36] Wikipedia. PyPy. http://en.wikipedia.org/wiki/PyPy, 2010.

[37] Wikipedia. Rubinius. http://en.wikipedia.org/wiki/Rubinius, 2010.

[38] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of PEPM'07, ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.

$$
\begin{array}{l}
\mathrm{U} = (* \to *) \to \\
\quad (* \to * \to *) \to \\
\quad (\Pi\chi : \Box.(\chi \to *) \to *) \to \\
\quad ((\Box \to *) \to *) \to \\
\quad *
\end{array}
$$

Figure 15. Kind of type representations

A. Encoding

A.1 Type Representations

In this section we provide concrete definitions of the kind U of type representations, the constructors of type representations, and the Fold[...] context. Our representations of types use a Church-style encoding. Figure 15 shows the kind U of type representations.

Theorem A.1. $\langle\rangle \vdash \mathrm{U} : \Box$

Proof. Straightforward. □

The file `lib/reptypes.pts` defines the constructors for our type representations.

Theorem A.2 (Kinds of Type Representation Constructors)**.**

$$
\begin{array}{l}
\langle\rangle \vdash \mathrm{Var} \quad : * \to \mathrm{U} \\
\langle\rangle \vdash \mathrm{Prod}_* \ : \mathrm{U} \to \mathrm{U} \to \mathrm{U} \\
\langle\rangle \vdash \mathrm{Prod}_\Box \ : (\Pi\chi : \Box.\ (\chi \to \mathrm{U}) \to \mathrm{U}) \\
\langle\rangle \vdash \mathrm{Prod}_\Delta \ : (\Box \to \mathrm{U}) \to \mathrm{U}
\end{array}
$$

Proof. Machine-checked. □

Definition A.1. *Suppose* var, prod_*, $\mathrm{prod}_\Box$, *and* prod_Δ *satisfy:*

$$
\begin{array}{l}
\langle\rangle \vdash \mathrm{var} \quad : * \to * \\
\langle\rangle \vdash \mathrm{prod}_* \ : * \to * \to * \\
\langle\rangle \vdash \mathrm{prod}_\Box \ : (\Pi\chi : \Box.\ (\chi \to *) \to *) \\
\langle\rangle \vdash \mathrm{prod}_\Delta \ : (\Box \to *) \to *
\end{array}
$$

Then $\mathrm{Fold}[\mathrm{var}, \mathrm{prod}_*, \mathrm{prod}_\Box, \mathrm{prod}_\Delta]$ *denotes the type:*

$$\lambda\alpha : \mathrm{U}.\ \alpha\ \mathrm{var}\ \mathrm{prod}_*\ \mathrm{prod}_\Box\ \mathrm{prod}_\Delta$$

Theorem A.3. *Let* $\mathrm{F} = \mathrm{Fold}[\mathrm{var}, \mathrm{prod}_*, \mathrm{prod}_\Box, \mathrm{prod}_\Delta]$. *Then* $\langle\rangle \vdash \mathrm{F} : \mathrm{U} \to *$

Proof. Straightforward. □

Theorem A.3 states that all folds on type representations have kind $\mathrm{U} \to *$. The proof is straightforward, since Fold[...] is only defined when the case functions have the expected kind.

Theorem A.4. *Suppose* $\Gamma \vdash \tau : *$, *and let* $\mathrm{F} = \mathrm{Fold}[\mathrm{F}_1, \mathrm{F}_2, \mathrm{F}_3, \mathrm{F}_4]$. *Then:*

$$
\begin{array}{ll}
\mathrm{F}\ \overline{\tau} \equiv_\beta \mathrm{F}_1\ \tau & \text{If } \tau \text{ is of the form } \alpha\ \tau_1\ \ldots\ \tau_n \\
\mathrm{F}\ \overline{\tau} \equiv_\beta \mathrm{F}_2\ (\mathrm{F}\ \overline{\tau_1})\ (\mathrm{F}\ \overline{\tau_2}) & \text{If } \tau = \tau_1 \to \tau_2,\ \Gamma \vdash \tau_1 : * \\
\mathrm{F}\ \overline{\tau} \equiv_\beta \mathrm{F}_3\ \kappa\ (\lambda\alpha : \kappa.\ \mathrm{F}\ \overline{\tau_1}) & \text{If } \tau = \Pi\alpha : \kappa.\tau_1,\ \Gamma \vdash \kappa : \Box \\
\mathrm{F}\ \overline{\tau} \equiv_\beta \mathrm{F}_4\ (\lambda\chi : \Box.\ \mathrm{F}\ \overline{\tau_1}) & \text{If } \tau = \Pi\chi : \Box.\ \tau_1
\end{array}
$$

Proof. By straightforward case analysis. In each case, we expand the definitions of $\overline{\tau}$, F, Fold[...], and the constructors Var, Prod$*$, $\mathrm{Prod}_\Box$, and Prod_Δ on both sides of the equivalence. Then a few simple β-reductions establish the equivalence. In the fifth case (when e is a type application of the form $\mathrm{e}_1\ \tau_2$, we also rely on Lemma A.11. □

$$
\begin{array}{l}
\mathrm{PExp} = \lambda\mathrm{R} : \mathrm{U} \to *.\ \lambda\alpha : \mathrm{U}. \\
\quad \mathrm{Witness\ R} \to \\
\quad \mathrm{Abs}_*\ \mathrm{R} \to \mathrm{App}_*\ \mathrm{R} \to \\
\quad \mathrm{Abs}_\Box\ \mathrm{R} \to \mathrm{App}_\Box\ \mathrm{R} \to \\
\quad \mathrm{Abs}_\Delta\ \mathrm{R} \to \mathrm{App}_\Delta\ \mathrm{R} \to \\
\quad \mathrm{R}\ \alpha
\end{array}
$$

Figure 16. Type of pre-quotations

Theorem A.4 states that an operation on type representations folds over the structure of its input. For example, a case function F_2 for $(*, *)$ products of the form $\tau_1 \to \tau_2$ maps the results of the operation on τ_1 and τ_2 to a result for $\tau_1 \to \tau_2$. The other cases are similar.

A.2 Term Representations

In this section we provide concrete definitions of the type PExp of pre-quotations, the constructors of term representations, and the fold[...] context. Our representations of types use a Church-style encoding. Figure 16 shows the type PExp of pre-quotations.

Theorem A.5. $\langle\rangle \vdash \mathrm{PExp} : (\mathrm{U} \to *) \to \mathrm{U} \to *$

Proof. Straightforward. □

The file `lib/replib.pts` defines the constructors for our term representations.

Theorem A.6 (Types of Term Representation Constructors)**.**

$$
\begin{array}{l}
\langle\rangle \vdash \mathrm{mkVar} \quad : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \mathrm{U}.\ \mathrm{R}\ \alpha \to \mathrm{PExp\ R}\ \alpha) \\
\\
\langle\rangle \vdash \mathrm{mkAbs}_* \ : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}. \\
\qquad (\mathrm{R}\ \alpha \to \mathrm{PExp\ R}\ \beta) \to \mathrm{PExp\ R}\ (\mathrm{Prod}_*\ \alpha\ \beta)) \\
\\
\langle\rangle \vdash \mathrm{mkApp}_* \ : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}. \\
\qquad \mathrm{PExp\ R}\ (\mathrm{Prod}_*\ \alpha\ \beta) \to \mathrm{PExp\ R}\ \alpha \to \mathrm{PExp\ R}\ \beta) \\
\\
\langle\rangle \vdash \mathrm{mkAbs}_\Box \ : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\kappa : \Box.\ \Pi\alpha : (\kappa \to \mathrm{U}). \\
\qquad (\Pi\beta : \kappa.\ \mathrm{PExp\ R}\ (\alpha\ \beta)) \to \mathrm{PExp\ R}\ (\mathrm{Prod}_\Box\ \kappa\ \alpha)) \\
\\
\langle\rangle \vdash \mathrm{mkApp}_\Box \ : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\kappa : \Box.\ \Pi\alpha : (\kappa \to \mathrm{U}). \\
\qquad \mathrm{PExp\ R}\ (\mathrm{Prod}_\Box\ \kappa\ \alpha) \to \Pi\beta : \kappa.\ \mathrm{PExp\ R}\ (\alpha\ \beta)) \\
\\
\langle\rangle \vdash \mathrm{mkAbs}_\Delta \ : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \Box \to \mathrm{U}. \\
\qquad (\Pi\chi : \Box.\ \mathrm{PExp\ R}\ (\alpha\ \chi)) \to \mathrm{PExp\ R}\ (\mathrm{Prod}_\Delta\ \alpha)) \\
\\
\langle\rangle \vdash \mathrm{mkApp}_\Delta \ : (\Pi\mathrm{R} : \mathrm{U} \to *.\ \Pi\alpha : \Box \to \mathrm{U}. \\
\qquad \mathrm{PExp\ R}\ (\mathrm{Prod}_\Delta\ \alpha) \to \Pi\chi : \Box.\ \mathrm{PExp\ R}\ (\alpha\ \chi))
\end{array}
$$

Proof. Machine-checked. □

Definition A.2. Suppose F, w, abs_*, app_*, $\mathrm{abs}_\Box$, $\mathrm{app}_\Box$, abs_Δ, and app_Δ satisfy:

$$
\begin{array}{lll}
\langle\rangle \vdash \mathrm{F} & : \mathrm{U} \to * & \\
\langle\rangle \vdash \mathrm{w} & : \mathrm{Witness\ F} & \\
\langle\rangle \vdash \mathrm{abs}_* & : \mathrm{Abs}_*\ \mathrm{F} & \langle\rangle \vdash \mathrm{app}_* \ : \mathrm{App}_*\ \mathrm{F} \\
\langle\rangle \vdash \mathrm{abs}_\Box & : \mathrm{Abs}_\Box\ \mathrm{F} & \langle\rangle \vdash \mathrm{app}_\Box \ : \mathrm{App}_\Box\ \mathrm{F} \\
\langle\rangle \vdash \mathrm{abs}_\Delta & : \mathrm{Abs}_\Delta\ \mathrm{F} & \langle\rangle \vdash \mathrm{app}_\Delta \ : \mathrm{App}_\Delta\ \mathrm{F}
\end{array}
$$

Then $\mathrm{fold}[\mathrm{F}, \mathrm{w}, \mathrm{abs}_*, \mathrm{app}_*, \mathrm{abs}_\Box, \mathrm{app}_\Box, \mathrm{abs}_\Delta, \mathrm{app}_\Delta]$ denotes the term:

$$\lambda\alpha : \mathrm{U}.\ \lambda\mathrm{e} : \mathrm{PExp\ F}\ \alpha.\ \mathrm{e\ w}\ \mathrm{abs}_*\ \mathrm{app}_*\ \mathrm{abs}_\Box\ \mathrm{app}_\Box\ \mathrm{abs}_\Delta\ \mathrm{app}_\Delta$$

Theorem A.7. Suppose F, w, abs_*, app_*, $\mathrm{abs}_\Box$, $\mathrm{app}_\Box$, abs_Δ, and app_Δ are as in Definition A.2. If $\mathrm{f} = \mathrm{fold}[\mathrm{F}, \mathrm{w}, \mathrm{abs}_*, \mathrm{app}_*, \mathrm{abs}_\Box, \mathrm{app}_\Box, \mathrm{abs}_\Delta, \mathrm{app}_\Delta]$, then $\Gamma \vdash \mathrm{f} : (\Pi\alpha : \mathrm{U}.\ \mathrm{PExp\ F}\ \alpha \to \mathrm{F}\ \alpha)$.

Proof. Straightforward. □

Pre-quotations are functions of seven arguments – a witness, and six fold functions. To reason about the behavior of a pre-quotation, we must consider applications of it to seven arguments. As a convenience, we define a one-hole context $\psi\langle\cdot\rangle$ to form such applications.

Definition A.3. *For any terms* e, w, $f_1, \ldots, f_6$ *(where* w, $f_1, \ldots, f_6$ *are to be inferred by context),* $\psi\langle e\rangle$ *denotes the term* e w f_1 ... f_6.

Lemma A.1 (Semantics of representation constructors)**.** *For any* w, $f_1, \ldots, f_6$*:*

1) $\psi\langle \text{mkVar R } \tau \text{ x}\rangle \equiv_\beta \text{x}$
2) $\psi\langle \text{mkAbs}_* \text{ R } \tau_1\ \tau_2 \text{ q}\rangle \equiv_\beta \text{f}_1\ \tau_1\ \tau_2\ (\lambda \text{x} : \text{R } \tau_1.\ \psi\langle \text{q x}\rangle)$
3) $\psi\langle \text{mkApp}_* \text{ R } \tau_1\ \tau_2\ \text{q}_1\ \text{q}_2\rangle \equiv_\beta \text{f}_2\ \tau_1\ \tau_2\ \psi\langle \text{q}_1\rangle\ \psi\langle \text{q}_2\rangle$
4) $\psi\langle \text{mkAbs}_\square \text{ R } \kappa\ \tau \text{ q}\rangle \equiv_\beta \text{f}_3\ \kappa\ \tau\ (\lambda\alpha : \kappa.\ \psi\langle \text{q}\rangle)$
5) $\psi\langle \text{mkApp}_\square \text{ R } \kappa\ \tau \text{ q } \tau_1\rangle \equiv_\beta \text{f}_4\ \kappa\ \tau\ \psi\langle \text{q}\rangle\ \tau_1$
6) $\psi\langle \text{mkAbs}_\Delta \text{ R } \tau \text{ q}\rangle \equiv_\beta \text{f}_5\ \tau\ (\lambda\chi : \square.\ \psi\langle \text{q}\rangle)$
7) $\psi\langle \text{mkApp}_\Delta \text{ R } \tau \text{ q } \kappa\rangle \equiv_\beta \text{f}_6\ \tau\ \psi\langle \text{q}\rangle\ \kappa$

Proof. Straightforward β-reduction. See the definitions in `lib/replib.pts`. □

While folds are only defined on prequotations, which have types of the form PExp F, we can apply a fold to a quotations by applying the quotation to the result type F.

Theorem A.8. *Suppose* F, w, abs_*, app_*, $\text{abs}_\square$, $\text{app}_\square$, abs_Δ, *and* app_Δ *are as in Definition A.2. If* $\text{f} = \textit{fold[F, w, abs}_*\textit{, app}_*\textit{, abs}_\square\textit{, app}_\square\textit{, abs}_\Delta\textit{, app}_\Delta\textit{], then:}$

$$\langle\rangle \vdash (\lambda\alpha : \text{U}.\ \lambda\text{x} : \text{Exp } \alpha.\ \text{f } \alpha\ (\text{x F})) : (\Pi\alpha : \text{U}.\ \text{Exp } \alpha \to \text{F } \alpha)$$

Proof. We have that $\text{Exp } \alpha \equiv_\beta (\Pi\text{R} : \text{U} \to *.\ \text{PExp R } \alpha)$, so $\alpha : \text{U}, \text{x} : \text{Exp } \alpha \vdash \text{x F} : \text{PExp F } \alpha$. By Theorem A.7 $\alpha : \text{U}, \text{x} : \text{Exp } \alpha \vdash f : (\Pi\alpha : \text{U}.\ \text{PExp F } \alpha \to \text{F}\alpha)$. Therefore, $\alpha : \text{U}.\ \text{x} : \text{Exp } \alpha \vdash \text{f } \alpha\ (\text{x F}) : \text{F } \alpha$. Therefore, we can derive the result by two uses of the abstraction rule. □

A.3 Coercions

The need for coercions arises from the fact that not every type of kind U is a type representation. Consider a type $\tau = \alpha \to \beta$ of kind $*$. We can construct multiple types of kind U from the components of τ:

$$\text{Var } (\alpha \to \beta)$$
$$\text{Prod}_*\ (\text{Var } \alpha)\ (\text{Var } \beta)$$

It happens that the second type is $\overline{\tau}$, the representation of τ. We might call the first a *pseudo-representation* of τ: it is a type of kind U, the kind of type representations, but is not the representation of τ.

A similar situation arises with term representations. Suppose a term e has type $\Pi\alpha : \kappa.\tau$, and a type τ_1 has kind κ. Then $e\ \tau_1$ has type $\tau[\alpha := \tau_1]$. Now, suppose $\overline{e}$ has type PExp R τ. Then $\text{mkApp}_\square\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau})\ \overline{e}\ \tau_1$ has the type $\text{PExp R } (\overline{\tau}[\alpha := \tau_1])$. This is in general not equivalent to $\text{PExp R } (\overline{\tau[\alpha := \tau_1]})$, so we insert a coercion to convert a term of type $\text{PExp R } (\overline{\tau}[\alpha := \tau_1])$ to one of type $\text{PExp R } (\overline{\tau[\alpha := \tau_1]})$. These coercions are automatically constructed by the quoter. To enable this, we require operations with a result type function R to satisfy the property:

Property A.1 (Coercibility of operations)**.** *For all types* τ *and* τ_1*, there exists a coercion* c *of type:* $\text{R } (\overline{\tau}[\alpha := \tau_1]) \to \text{R } (\overline{\tau[\alpha := \tau_1]})$.

The property is witnessed by a tuple of functions that the quoter uses to construct coercions. There are three coercion functions for each of the rules $(*, *)$, $(\square, *)$, and $(\Delta, *)$. They are:

dist_s	Adds one level of type representation.
factor_s	Removes one level of type representation.
coerce_s	Applies coercion(s) to the components of a representation.

When we say that dist_s adds a level of type representation, we mean that it encodes only the top-level form of a type. For example, dist_* maps terms with types of the form $\text{R } (\text{Var } (\alpha \to \beta))$ to $\text{R } (\text{Prod}_*\ (\text{Var } \alpha)\ (\text{Var } \beta))$. The name dist_* reflects that we distribute Var over the arrow.

The factor_s witness functions go the opposite direction as the dist_s ones. For example, factor_* maps terms with types of the form $\text{R } (\text{Prod}_*\ (\text{Var } \alpha)\ (\text{Var } \beta))$ to $\text{R } (\text{Var } (\alpha \to \beta))$.

The coerce_s witness functions is a constructor for coercions on $(s, *)$ products.

We use the witness functions to define nine coercion functions with types listed in Figure 18. Coercions of type Reify_* R map $\text{R } (\text{Var } (\tau_1 \to \tau_2))$ terms to $\text{R } \overline{\tau_1 \to \tau_2}$ terms. They take as input a reflect coercion for τ_1 and a reify coercion for τ_2. Coercions of type Reflect_* R map $\text{R } \overline{\tau_1 \to \tau_2}$ terms to $\text{R } (\text{Var } (\tau_1 \to \tau_2))$ terms. They take as input a reify coercion for τ_1 and a reflect coercion for τ_2. Coercions of type Coerce_* R map $\text{R } (\text{Prod}_*\ \tau_1\ \tau_2)$ terms to $\text{R } (\text{Prod}_*\ \sigma_1\ \sigma_2)$ terms, taking as input coercions from $\text{R } \sigma_1$ to $\text{R } \tau_1$ and from $\text{R } \tau_2$ to $\text{R } \sigma_2$.

Each operation is required to supply a witness of property A.1, which is a tuple of nine functions. The types of these functions are listed in Figure 17.

A.4 Coercion Constructors

We use witnesses to define coercion constructors, whose types are listed in Figure 18. The quoter uses the constructors to build coercions with types of the form

$$\text{PExp R } (\overline{\tau}[\alpha := \tau_1]) \to \text{PExp R } (\overline{\tau[\alpha := \tau_1]})$$

The file `lib/coercelib.pts` defines the functions reify_s, reflect_s, and coerce_s for each $s \in \{*, \square, \Delta\}$.

Lemma A.2. *The functions* reify_s*,* reflect_s*, and* coerce_s *for each* $s \in \{*, \square, \Delta\}$ *have the types listed in Figure 18.*

Proof. Machine checked. □

Figure 19 defines the coercion construction process used by the quoter. The notation $\tau_1 \rightsquigarrow \tau_2$ denotes the coercion from PExp R τ_1 terms to PExp R τ_2 terms. They are constructed inductively by the structure of τ_1 and τ_2.

Lemma A.3 (Reification and Reflection)**.** *If* τ *is a normal form and* $\Gamma \vdash \tau : *$*, then there exist terms* c_1 *and* c_2 *such that:*

$$\text{Var } \tau \rightsquigarrow \overline{\tau} = c_1$$
$$\overline{\tau} \rightsquigarrow \text{Var } \tau = c_2$$

Proof. Straightforward, by induction on the height of the derivation of $\Gamma \vdash \tau : *$. □

Theorem A.9 (Completeness of Coercion Construction)**.** *If* $\Gamma, \alpha : \kappa \vdash \tau : *$ *and* $\Gamma \vdash \tau_1 : *$*, then there exists terms* c_1 *and* c_2 *such that:*

$$\overline{\tau}[\alpha := \tau_1] \rightsquigarrow \overline{\tau[\alpha := \tau_1]} = c_1$$
$$\overline{\tau[\alpha := \tau_1]} \rightsquigarrow \overline{\tau}[\alpha := \tau_1] = c_2$$

Proof. Straightforward by induction on the height of $\Gamma, \alpha : \kappa \vdash \tau : *$. □

```
Dist* : (U → *) → * =
  λR : U → *. Πα:U. Πβ:U.
  R (Var (UId α → UId β)) →
  R (Prod* (Var (UId α)) (Var (UId β)))

Factor* : (U → *) → * =
  λR : U → *. Πα:U. Πβ:U.
  R (Prod* (Var (UId α)) (Var (UId β))) →
  R (Var (UId α → UId β))

Coerce* : (U → *) → * =
  λR : U → *. Πα:U. Πβ:U. Πα1:U. Πβ1:U.
  (R α1 → R α ) → (R β → R β1) →
  R (Prod* α β) → R (Prod* α1 β1)

Dist□ : (U → *) → * =
  λR : U → *. Πχ:□. Πα:χ → U.
  R (Var (Πβ:χ. UId (α β))) →
  R (Prod□ χ (λβ:χ. Var (UId (α β))))

Factor□ : (U → *) → * =
  λR : U → *. Πχ:□. Πα:χ → U.
  R (Prod□ χ (λβ:χ. Var (UId (α β)))) →
  R (Var (Πβ:χ. UId (α β)))

Coerce□ : (U → *) → * =
  λR : U → *. Πχ:□. Πα1: χ → U. Πα2: χ → U.
  (Πβ:χ. R (α1 β) → R (α2 β)) →
  R (Prod□ χ α1) → R (Prod□ χ α2)

Dist△ : (U → *) → * =
  λR : U → *. Πα : □ → U.
  R (Var (Πχ:□. UId (α χ))) →
  R (Prod△ (λχ:□. Var (UId (α χ))))

Factor△ : (U → *) → * =
  λR : U → *. Πα:□ → U.
  R (Prod△ (λχ:□. Var (UId (α χ)))) →
  R (Var (Πχ:□. UId (α χ)))

Coerce△ : (U → *) → * =
  λR : U → *. Πα1: □ → U. Πα2: □ → U.
  (Πχ:□. R (α1 χ) → R (α2 χ)) →
  R (Prod△ α1) → R (Prod△ α2)

Witness : (U → *) → * =
  λR : U → *. Πα : *.
  (Dist* R → Dist□ R → Dist△ R →
   Factor* R → Factor□ R → Factor△ R →
   Coerce* R → Coerce□ R → Coerce△ R →
   α) →
  α
```

Figure 17. Types of Witness Functions

$$
\begin{aligned}
\text{Reify}_* &: \lambda R : U \to *.\ \Pi\alpha : U.\ \Pi\beta : U. \\
&\quad (R\ \alpha \to R\ (\text{Var}\ (\text{UId}\ \alpha))) \to \\
&\quad (R\ (\text{Var}\ (\text{UId}\ \beta)) \to R\ \beta) \to \\
&\quad R\ (\text{Var}\ (\text{UId}\ \alpha \to \text{UId}\ \beta)) \to R\ (\text{Prod}_*\alpha\ \beta) \\
\text{Reflect}_* &: \lambda R : U \to *.\ \Pi\alpha : U.\ \Pi\beta : U. \\
&\quad (R\ (\text{Var}\ (\text{UId}\ \alpha)) \to R\ \alpha) \to \\
&\quad (R\ \beta \to R\ (\text{Var}\ (\text{UId}\ \beta))) \to \\
&\quad R\ (\text{Prod}_*\alpha\ \beta) \to R\ (\text{Var}\ (\text{UId}\ \alpha \to \text{UId}\ \beta)) \\
\\
\text{Reify}_\square &: \lambda R : U \to *.\ \Pi\chi : \square.\ \Pi\tau : \chi \to U. \\
&\quad (\Pi\alpha : \chi.\ R\ (\text{Var}\ (\text{UId}\ (\tau\ \alpha))) \to R\ (\tau\ \alpha)) \to \\
&\quad R\ (\text{Var}\ (\Pi\alpha : \chi.\ \text{UId}\ (\tau\ \alpha))) \to R\ (\text{Prod}_\square\ \kappa\ \tau) \\
\text{Reflect}_\square &: \lambda R : U \to *.\ \Pi\chi : \square.\ \Pi\tau : \chi \to U. \\
&\quad (\Pi\alpha : \chi.\ R\ (\tau\ \alpha) \to R\ (\text{Var}\ (\text{UId}\ (\tau\ \alpha)))) \to \\
&\quad R\ (\text{Prod}_\square\ \kappa\ \tau) \to R\ (\text{Var}\ (\Pi\alpha : \chi.\ \text{UId}\ (\tau\ \alpha))) \\
\\
\text{Reify}_\triangle &: \lambda R : U \to *.\ \Pi\tau : \square \to U. \\
&\quad (\Pi\chi : \square.\ R\ (\text{Var}\ (\text{UId}\ (\tau\ \chi))) \to R\ (\tau\ \chi)) \to \\
&\quad R\ (\text{Var}\ (\Pi\chi : \square.\ \text{UId}\ (\tau\ \chi))) \to R\ (\text{Prod}_\triangle\ \tau) \\
\text{Reflect}_\triangle &: \lambda R : U \to *.\ \Pi\tau : \square \to U. \\
&\quad (\Pi\chi : \square.\ R\ (\tau\ \chi) \to R\ (\text{Var}\ (\text{UId}\ (\tau\ \chi)))) \to \\
&\quad R\ (\text{Prod}_\triangle\ \tau) \to R\ (\text{Var}\ (\Pi\chi : \square.\ \text{UId}\ (\tau\ \chi))) \\
\\
\text{reify}_* &: \text{Reify}_*\ (\text{PExp R}) \\
\text{reflect}_* &: \text{Reflect}_*\ (\text{PExp R}) \\
\text{coerce}_* &: \text{Coerce}_*\ (\text{PExp R}) \\
\text{reify}_\square &: \text{Reify}_\square\ (\text{PExp R}) \\
\text{reflect}_\square &: \text{Reflect}_\square\ (\text{PExp R}) \\
\text{coerce}_\square &: \text{Coerce}_\square\ (\text{PExp R}) \\
\text{reify}_\triangle &: \text{Reify}_\triangle\ (\text{PExp R}) \\
\text{reflect}_\triangle &: \text{Reflect}_\triangle\ (\text{PExp R}) \\
\text{coerce}_\triangle &: \text{Coerce}_\triangle\ (\text{PExp R})
\end{aligned}
$$

Figure 18. Types of Coercion Functions

Theorem A.10 (Types of Coercions)**.** *If* $\Gamma \vdash \tau_1 : \text{U}$ *and* $\Gamma \vdash \tau_2 : \text{U}$*, and* $\tau_1 \rightsquigarrow \tau_2 = c$*, then*

$$R : \text{U} \to *, \overline{\Gamma} \vdash c : \text{PExp R}\ \tau_1 \to \text{PExp R}\ \tau_2$$

Proof. Straightforward by induction on the height of $\tau_1 \rightsquigarrow \tau_2 = c$, and by Lemma A.2 and Theorem 4.2. □

Now we turn to the semantics of coercions. Coercions are built from witnesses, which Church tuples of 9 components. We begin by defining a set of witness projection functions.

Definition A.4 (Witness Projection Functions)**.** *Let R be a type of kind* $\text{U} \to *$*, which is to be inferred from context. For* $i \in \{1, \ldots, 9\}$*,* w_i *denotes the witness projection function:*

$$
\begin{aligned}
&\lambda x_1 : \text{Dist}_*\ R.\ \ \lambda x_2 : \text{Factor}_*\ R.\ \ \lambda x_3 : \text{Coerce}_*\ R. \\
&\lambda x_4 : \text{Dist}_\square\ R.\ \ \lambda x_5 : \text{Factor}_\square\ R.\ \ \lambda x_6 : \text{Coerce}_\square\ R. \\
&\lambda x_7 : \text{Dist}_\triangle\ R.\ \ \lambda x_8 : \text{Factor}_\triangle\ R.\ \ \lambda x_9 : \text{Coerce}_\triangle\ R.\ x_i
\end{aligned}
$$

Lemma A.4 (Witness Projections are Normal Forms)**.** *For* $i \in \{1, \ldots, 9\}$*,* w_i *is a normal form.*

Proof. Trivial. □

Now we turn to the semantics of coercions. Coercions rely on a function lift that maps terms of type $\text{R}\ \alpha$ to terms of type $\text{PExp R}\ \alpha$. The definition of lift is in `lib/replib.pts`.

Lemma A.5 (Type of lift)**.** $\langle\rangle \vdash \text{lift} : (\Pi R : \text{U} \to *.\ \Pi\alpha : \text{U}.\ \text{R}\ \alpha \to \text{PExp R}\ \alpha)$

$$\frac{}{(\text{Var } \tau \rightsquigarrow \text{Var } \tau) = \lambda x : \text{PExp R (Var } \tau).x}$$

$$\frac{(\overline{\tau_1} \rightsquigarrow \text{Var } \tau_1) = \text{reflect}_{\tau_1} \qquad (\text{Var } \tau_2 \rightsquigarrow \overline{\tau_2}) = \text{reify}_{\tau_2}}{(\text{Var } (\tau_1 \to \tau_2) \rightsquigarrow \overline{\tau_1 \to \tau_2}) = \text{reify}_* \text{ R } \overline{\tau_1}\ \overline{\tau_2} \text{ reflect}_{\tau_1} \text{ reify}_{\tau_2}}$$

$$\frac{(\text{Var } \tau_1 \rightsquigarrow \overline{\tau_1}) = \text{reify}_{\tau_1} \qquad (\overline{\tau_2} \rightsquigarrow \text{Var } \tau_2) = \text{reflect}_{\tau_2}}{(\overline{\tau_1 \to \tau_2} \rightsquigarrow \text{Var } (\tau_1 \to \tau_2)) = \text{reflect}_* \text{ R } \overline{\tau_1}\ \overline{\tau_2} \text{ reify}_{\tau_1} \text{ reflect}_{\tau_2}}$$

$$\frac{(\sigma_1 \rightsquigarrow \tau_1) = \text{c}_1 \qquad (\tau_2 \rightsquigarrow \sigma_2) = \text{c}_2}{(\text{Prod}_* \ \tau_1\ \tau_2 \rightsquigarrow \text{Prod}_* \ \sigma_1\ \sigma_2) = \text{coerce}_* \text{ R } \tau_1\ \tau_2\ \sigma_1\ \sigma_2\ \text{c}_1\ \text{c}_2}$$

$$\frac{(\text{Var } \tau) \rightsquigarrow \overline{\tau} = \text{c}}{\text{Var } (\Pi\alpha : \kappa.\tau) \rightsquigarrow \overline{\Pi\alpha : \kappa.\tau} = \text{reify}_\square \text{ R } \kappa\ (\lambda\alpha : \kappa.\overline{\tau})\ (\lambda\alpha : \kappa.\ \text{c})}$$

$$\frac{\overline{\tau} \rightsquigarrow (\text{Var } \tau) = \text{c}}{\overline{\Pi\alpha : \kappa.\tau} \rightsquigarrow \text{Var } (\Pi\alpha : \kappa.\tau) = \text{reflect}_\square \text{ R } \kappa\ (\lambda\alpha : \kappa.\overline{\tau})\ (\lambda\alpha : \kappa.\ \text{c})}$$

$$\frac{\tau \rightsquigarrow \sigma = \text{c}}{\begin{array}{c}(\text{Prod}_\square\ \kappa\ (\lambda\alpha : \kappa.\ \tau)) \rightsquigarrow (\text{Prod}_\square\ \kappa\ (\lambda\alpha : \kappa.\ \sigma)) \\ = \text{coerce}_\square \text{ R } \kappa\ (\lambda\alpha : \kappa.\tau)\ (\lambda\alpha : \kappa.\sigma)\ (\lambda\alpha : \kappa.\ \text{c})\end{array}}$$

$$\frac{(\text{Var } \tau) \rightsquigarrow \overline{\tau} = \text{c}}{\text{Var } (\Pi\chi : \square.\tau) \rightsquigarrow \overline{\Pi\chi : \square.\tau} = \text{reify}_\Delta \text{ R } (\lambda\chi : \square.\overline{\tau})\ (\lambda\chi : \square.\ \text{c})}$$

$$\frac{\overline{\tau} \rightsquigarrow (\text{Var } \tau) = \text{c}}{\overline{\Pi\chi : \square.\tau} \rightsquigarrow \text{Var } (\Pi\chi : \square.\tau) = \text{reflect}_\Delta \text{ R } (\lambda\chi : \square.\overline{\tau})\ (\lambda\chi : \square.\ \text{c})}$$

$$\frac{\tau \rightsquigarrow \sigma = \text{c}}{\begin{array}{c}(\text{Prod}_\Delta\ (\lambda\chi : \square.\ \tau)) \rightsquigarrow (\text{Prod}_\Delta\ (\lambda\chi : \square.\ \sigma)) \\ = \text{coerce}_\Delta \text{ R } (\lambda\chi : \square.\tau)\ (\lambda\chi : \square.\sigma)\ (\lambda\chi : \square.\ \text{c})\end{array}}$$

Figure 19. Coercion Construction

Proof. Machine-checked. □

Lemma A.6 (Semantics of lift)**.** *For all* R, α, e, $\psi\langle\text{lift R } \alpha\ \text{e}\rangle \equiv_\beta \text{e}$

Proof. Straightforward β-reduction. See the definition of lift in `lib/replib.pts`. □

Lemma A.7 (Semantics of primitive coercion constructors)**.** *For any* w, $\text{f}_1, \ldots, \text{f}_6$*:*

1) $\psi\langle\text{dist}_*\ \text{R}\ \tau_1\ \tau_2\ \text{e}\rangle \equiv_\beta \text{w (Dist}_*\ \text{R) w}_1\ \tau_1\ \tau_2\ \psi\langle\text{e}\rangle$

2) $\psi\langle\text{factor}_*\ \text{R}\ \tau_1\ \tau_2\ \text{e}\rangle \equiv_\beta \text{w (Factor}_*\ \text{R) w}_2\ \tau_1\ \tau_2\ \psi\langle\text{e}\rangle$

3) $\psi\langle\text{coerce}_*\ \text{R}\ \tau_1\ \tau_2\ \sigma_1\ \sigma_2\ \text{c}_1\ \text{c}_2\ \text{e}\rangle \equiv_\beta \text{w (Coerce}_*\ \text{R) w}_3\ \tau_1\ \tau_2\ \sigma_1\ \sigma_2\ (\lambda x : \text{R } \sigma_1.\ \psi\langle c_1\ (\text{lift R } \sigma_1\ x)\rangle)\ (\lambda x : \text{R } \tau_2.\ \psi\langle c_2\ (\text{lift R } \tau_2\ x)\rangle)\ \psi\langle\text{e}\rangle$

4) $\psi\langle\text{dist}_\square\ \text{R}\ \kappa\ \tau\ \text{e}\rangle \equiv_\beta \text{w (Dist}_\square\ \text{R) w}_4\ \kappa\ \tau\ \psi\langle\text{e}\rangle$

5) $\psi\langle\text{factor}_\square\ \text{R}\ \kappa\ \tau\ \text{e}\rangle \equiv_\beta \text{w (Factor}_\square\ \text{R) w}_5\ \kappa\ \tau\ \psi\langle\text{e}\rangle$

6) $\psi\langle\text{coerce}_\square\ \text{R}\ \kappa\ \tau\ \sigma\ \text{c e}\rangle \equiv_\beta \text{w (Coerce}_\square\ \text{R) w}_6\ \kappa\ \tau\ \sigma\ (\lambda\alpha : \kappa.\ \lambda x : \text{R } (\tau\ \alpha).\ \psi\langle\text{c } \alpha\ (\text{lift R } (\tau\ \alpha)\ x)\rangle)\ \psi\langle\text{e}\rangle$

7) $\psi\langle\text{dist}_\Delta\ \text{R}\ \tau\ \text{e}\rangle \equiv_\beta \text{w (Dist}_\Delta\ \text{R) w}_7\ \kappa\ \tau\ \psi\langle\text{e}\rangle$

8) $\psi\langle\text{factor}_\Delta\ \text{R}\ \tau\ \text{e}\rangle \equiv_\beta \text{w (Factor}_\Delta\ \text{R) w}_8\ \tau\ \psi\langle\text{e}\rangle$

9) $\psi\langle\text{coerce}_\Delta\ \text{R}\ \tau\ \sigma\ \text{c e}\rangle \equiv_\beta \text{w (Coerce}_\Delta\ \text{R) w}_9\ \tau\ \sigma\ (\lambda\chi : \square.\ \lambda x : \text{R } (\tau\ \chi).\ \psi\langle\text{c } \chi\ (\text{lift R } (\tau\ \chi)\ x)\rangle)\ \psi\langle\text{e}\rangle$

Proof. Straightforward β-reduction. See the definitions in `lib/coercelib.pts`. □

Lemma A.8 (Semantics of derived coercion constructors)**.** *For any* w, $\text{f}_1, \ldots, \text{f}_6$*:*

1) $\psi\langle\text{reify}_*\ \text{R}\ \tau_1\ \tau_2\ \text{c}_1\ \text{c}_2\ \text{e}\rangle \equiv_\beta \psi\langle\text{coerce}_*\ \text{R (Var (UId } \tau_1\text{)) (Var (UId } \tau_2\text{))}\ \tau_1\ \tau_2\ \text{c}_1\ \text{c}_2\ (\text{dist}_*\ \text{R}\ \tau_1\ \tau_2\ \text{e})\rangle$

2) $\psi\langle\text{reflect}_*\ \text{R}\ \tau_1\ \tau_2\ \text{c}_1\ \text{c}_2\ \text{e}\rangle \equiv_\beta \psi\langle\text{factor}_*\ \text{R}\ \tau_1\ \tau_2\ (\text{coerce}_*\ \text{R}\ \tau_1\ \tau_2\ (\text{Var (UId } \tau_1\text{)) (Var (UId } \tau_2\text{))}\ \text{c}_1\ \text{c}_2\ \text{e})\rangle$

3) $\psi\langle\text{reify}_\square\ \text{R}\ \kappa\ \tau\ \text{c e}\rangle \equiv_\beta \psi\langle\text{coerce}_\square\ \text{R}\ \kappa\ (\lambda\alpha : \kappa.\ \text{Var (UId } (\tau\ \alpha))\ \tau\ \text{c}\ (\text{dist}_\square\ \text{R}\ \kappa\ \tau\ \text{e})\rangle$

4) $\psi\langle\text{reflect}_\square\ \text{R}\ \kappa\ \tau\ \text{c e}\rangle \equiv_\beta \psi\langle\text{factor}_\square\ \text{R}\ \kappa\ \tau\ (\text{coerce}_\square\ \text{R}\ \kappa\ \tau\ (\lambda\alpha : \kappa.\ \text{Var (UId } (\tau\ \alpha))\ \text{c e})\rangle$

5) $\psi\langle\text{reify}_\Delta\ \text{R}\ \tau\ \text{c e}\rangle \equiv_\beta \psi\langle\text{coerce}_\Delta\ \text{R}\ (\lambda\chi : \square.\ \text{Var (UId } (\tau\ \chi)))\ \tau\ \text{c}\ (\text{dist}_\Delta\ \text{R}\ \tau\ \text{e})\rangle$

6) $\psi\langle\text{reflect}_\Delta\ \text{R}\ \tau\ \text{c e}\rangle \equiv_\beta \psi\langle\text{factor}_\Delta\ \text{R}\ \tau\ (\text{coerce}_\Delta\ \text{R}\ \tau\ (\lambda\chi : \square.\ \text{Var (UId } (\tau\ \chi)))\ \text{c e})\rangle$

Proof. Straightforward β-reduction. See the definitions in `lib/coercelib.pts`. □

Lemmas A.9, A.10, and A.11 show that operating on a coerced pre-quotation is equivalent to operating on the prequotation, then coercing the result.

Lemma A.9.

1) $\psi\langle \text{dist}_* \text{ R } \tau_1 \ \tau_2 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
2) $\psi\langle \text{factor}_* \text{ R } \tau_1 \ \tau_2 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
3) $\psi\langle \text{coerce}_* \text{ R } \tau_1 \ \tau_2 \ \sigma_1 \ \sigma_2 \text{ c}_1 \text{ c}_2 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
4) $\psi\langle \text{dist}_\square \text{ R } \kappa \ \tau \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
5) $\psi\langle \text{factor}_\square \text{ R } \kappa \ \tau \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
6) $\psi\langle \text{coerce}_\square \text{ R } \kappa \ \tau \ \sigma \text{ c}_1 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
7) $\psi\langle \text{dist}_\Delta \text{ R } \kappa \ \tau \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
8) $\psi\langle \text{factor}_\Delta \text{ R } \kappa \ \tau \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
9) $\psi\langle \text{coerce}_\Delta \text{ R } \kappa \ \tau \ \sigma \text{ c}_1 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c

Proof. Straightforward by Lemma A.7. □

Lemma A.10.

1) $\psi\langle \text{reify}_* \text{ R } \tau_1 \ \tau_2 \text{ c}_1 \text{ c}_2 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
2) $\psi\langle \text{reflect}_* \text{ R } \tau_1 \ \tau_2 \text{ c}_1 \text{ c}_2 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
3) $\psi\langle \text{reify}_\square \text{ R } \kappa \ \tau \text{ c}_1 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
4) $\psi\langle \text{reflect}_\square \text{ R } \kappa \ \tau \text{ c}_1 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
5) $\psi\langle \text{reify}_\Delta \text{ R } \tau \text{ c}_1 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c
6) $\psi\langle \text{reflect}_\Delta \text{ R } \tau \text{ c}_1 \text{ e}\rangle \equiv_\beta \text{ c } \psi\langle \text{e}\rangle$, for some c

Proof. Straightforward by Lemma A.8 and Lemma A.9. Each case is similar, and (1) is representative.

Case (1): By Lemma A.8 and then Lemma A.9 (1) and (3), we have that:

$$\begin{aligned} & \psi\langle \text{reify}_* \text{ R } \tau_1 \ \tau_2 \text{ c}_1 \text{ c}_2 \text{ e}\rangle \\ \equiv_\beta \ & \psi\langle \text{coerce}_* \text{ R } (\text{Var } (\text{UId } \tau_1)) \ (\text{Var } (\text{UId } \tau_2)) \ \tau_1 \ \tau_2 \\ & \quad \text{c}_1 \text{ c}_2 \ (\text{dist}_* \text{ R } \tau_1 \ \tau_2 \text{ e})\rangle \\ \equiv_\beta \ & \text{c}' \ (\text{c}'' \ \psi\langle \text{e}\rangle) \end{aligned}$$

The result holds with $\text{c} = \lambda \text{x} : \text{R } (\text{Var } (\text{UId } \tau_1 \to \text{UId } \tau_2)).\ \text{c}' \ (\text{c}'' \ \text{x})$. □

Lemma A.11. *If $\tau_1 \rightsquigarrow \tau_2 = \text{c}$, then there exists a c' such that $\phi\langle \text{c e}\rangle \equiv_\beta \text{c}' \ \phi\langle \text{e}\rangle$ for all* e.

Proof. Straightforward, by Lemmas A.9 and A.10. □

The following theorem states that our encoding meets the specification of Definition 5.3.

Theorem A.11. Suppose F, w, abs_*, app_*, $\text{abs}_\square$, $\text{app}_\square$, abs_Δ, and app_Δ are as in Definition A.2, and suppose $\text{f} = \text{fold}[\text{F, w, abs}_*, \text{app}_*, \text{abs}_\square, \text{app}_\square, \text{abs}_\Delta, \text{app}_\Delta]$. Then for any context Γ, term e, and type τ such that $\Gamma \vdash \text{e} : \tau$, we have that:

If e is a variable, then $\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta \text{e}$.
If $\tau = \tau_1 \to \tau_2,\ \Gamma \vdash \tau_1 : *,$ and $\text{e} = \lambda \text{x} : \tau_1.\ \text{e}_1$, then
$\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta \text{abs}_* \ \overline{\tau_1} \ \overline{\tau_2} \ (\lambda \text{x} : \text{F } \overline{\tau_1}.\ \text{f } \overline{\tau_2} \ \overline{\text{e}_1})$.
If $\text{e} = \text{e}_1 \ \text{e}_2,\ \Gamma \vdash \text{e}_2 : \tau_1 : *$, then
$\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta \text{app}_* \ \overline{\tau_1} \ \overline{\tau} \ (\text{f } \overline{\tau_1 \to \tau} \ \overline{\text{e}_1}) \ (\text{f } \overline{\tau_1} \ \overline{\text{e}_2})$.
If $\tau = \Pi \alpha : \kappa.\ \tau_1,\ \Gamma \vdash \kappa : \square,$ and $\text{e} = \lambda \alpha : \kappa.\ \text{e}_1$, then
$\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta \text{abs}_\square \ \kappa \ (\lambda \alpha : \kappa.\ \overline{\tau_1}) \ (\lambda \alpha : \kappa.\ \text{f } \overline{\tau_1} \ \overline{\text{e}_1})$.
If $\text{e} = \text{e}_1 \ \tau_2,\ \Gamma \vdash \kappa : \square,$ and $\Gamma \vdash \text{e}_1 : \Pi \alpha : \kappa.\ \tau_1$, then
$\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta c \ (\text{app}_\square \ \kappa \ (\lambda \alpha : \kappa.\ \overline{\tau_1}) \ (\text{f } \overline{\Pi \alpha : \kappa.\ \tau_1} \ \overline{\text{e}_1}) \ \tau_2)$
for some coercion c
If $\tau = \Pi \chi : \square.\ \tau_1,$ and $\text{e} = \lambda \chi : \square.\ \text{e}_1$, then
$\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta \text{abs}_\Delta \ (\lambda \chi : \square . \overline{\tau_1}) \ (\lambda \chi : \square.\ \text{f } \overline{\tau_1} \ \overline{\text{e}_1})$.
If $\text{e} = \text{e}_1 \ \kappa,\ \Gamma \vdash \kappa : \square,$ and $\Gamma \vdash \text{e}_1 : \Pi \chi : \square.\ \tau_1$, then
$\text{f } \overline{\tau} \ \overline{\text{e}} \equiv_\beta \text{app}_\Delta \ (\lambda \chi : \square.\ \overline{\tau_1}) \ (\text{f } \overline{\Pi \chi : \square . \tau_1} \ \overline{\text{e}_1}) \ \kappa$.

Proof. By straightforward case analysis. For each case, we expand the definitions of $\overline{\tau}$, $\overline{\text{e}}$, f, fold$[\dots]$ on both sides of the equivalence, and apply Lemma A.1. For the fifth case (type application), we use Lemma A.11 □

B. Normalization of Representations

We now prove strong normalization for our representations. For convenience, we define a new one-hole context $\theta\langle\cdot\rangle$, which is similar to $\psi\langle\cdot\rangle$ except that we require that w, $\text{f}_1, \dots, \text{f}_6$ be variables.

Definition B.1. *For any term* e, $\theta\langle \text{e}\rangle$ *denotes the term* $\text{e w f}_1 \ \dots \ \text{f}_6$, *where w, $f_1, \dots, f_6$ are variables.*

Definition B.2. *Suppose that $\theta\langle \text{e}\rangle$ is strongly normalizing (SN). Then we say that* e *is θ-SN.*

Definition B.3. *Suppose that* c *is a term such that whenever* e *is θ-SN,* c e *is θ-SN. Then we say that* c *is θ2-SN.*

Lemma B.1. *If* e *is θ-SN, then* e *is SN.*

Proof. Suppose e is θ-SN. By definition, $\theta\langle \text{e}\rangle \equiv_\beta \text{e w f}_1 \ \dots \ \text{f}_6$ is SN. Therefore, e is SN. □

Lemma B.2. *For any* R, τ, e, *if* e *is SN, then* $\text{lift R } \tau \text{ e}$ *is θ-SN.*

Proof. Suppose e is SN. We must show that $\theta\langle \text{lift R } \tau \ e\rangle$ is SN. We have that $\theta\langle\cdot\rangle$ is a particular case of a $\psi\langle\cdot\rangle$, so by Lemma A.6 we have that $\theta\langle \text{lift R } \tau \text{ e}\rangle \equiv_\beta \text{e}$. By assumption, e is SN, so $\theta\langle \text{lift R } \tau \text{ e}\rangle$ is SN as required. □

Lemma B.3.

1. $\forall \text{R}, \tau_1, \tau_2$: $\text{dist}_* \text{ R } \tau_1 \ \tau_2$ *is θ2-SN.*
2. $\forall \text{R}, \tau_1, \tau_2$: $\text{factor}_* \text{ R } \tau_1 \ \tau_2$ *is θ2-SN.*
3. $\forall \text{R}, \tau_1, \tau_2, \sigma_1, \sigma_2, \text{c}_1, \text{c}_2$: *If* c_1 *and* c_2 *are θ2-SN, then* $\text{coerce}_* \text{ R } \tau_1 \ \tau_2 \ \sigma_1 \ \sigma_2 \text{ c}_1 \text{ c}_2$ *is θ2-SN.*
4. $\forall \text{R}, \kappa, \tau$: $\text{dist}_\square \text{ R } \kappa \ \tau$ *is θ2-SN.*
5. $\forall \text{R}, \kappa, \tau$: $\text{factor}_\square \text{ R } \kappa \ \tau$ *is θ2-SN.*
6. $\forall \text{R}, \kappa, \tau, \sigma, \text{c}$: *If* c α *is θ2-SN for all α, then* $\text{coerce}_\square \text{ R } \kappa \ \tau \ \sigma \text{ c}$ *is θ2-SN.*
7. $\forall \text{R}, \tau$: $\text{dist}_\Delta \text{ R } \kappa \ \tau$ *is θ2-SN.*
8. $\forall \text{R}, \tau$: $\text{factor}_\Delta \text{ R } \kappa \ \tau$ *is θ2-SN.*
9. $\forall \text{R}, \tau, \sigma, \text{c}$: *If* c χ *is θ2-SN for all χ, then* $\text{coerce}_\Delta \text{ R } \tau \ \sigma \text{ c}$ *is θ2-SN.*

Proof. Each case is similar. (9) is representative.

Suppose c χ is θ2-SN for all χ, and suppose e is θ-SN. We must show that $\theta\langle \text{coerce}_\Delta \text{ R } \tau \ \sigma \text{ c e}\rangle$ is SN. By Lemma A.7, we have that:

$$\begin{aligned} & \theta\langle \text{coerce}_\Delta \text{ R } \tau \ \sigma \text{ c e}\rangle \\ \equiv_\beta \ & \text{w } (\text{Coerce}_\Delta \text{ R}) \ \text{w}_9 \ \tau \ \sigma \\ & \quad (\lambda \chi : \square.\ \lambda \text{x} : \text{R } (\tau \ \chi).\ \theta\langle \text{c } \chi \ (\text{lift R } (\tau \ \chi) \ \text{x})\rangle) \ \theta\langle \text{e}\rangle \end{aligned}$$

Note that w is a variable. By Lemma A.4, we have that w_9 is a normal form. Since $(\text{Coerce}_\Delta \text{ R})$, τ, and σ are types, they are all SN. Since x is a variable, it is SN, so $(\text{lift R } (\tau \ \chi) \ \text{x})$ is θ-SN by Lemma B.2. Since c χ is θ2-SN, $\theta\langle \text{c } \chi \ (\text{lift R } (\tau \ \chi) \ \text{x})\rangle$ is SN. Therefore the term $(\lambda \chi : \square.\ \dots)$ is SN. Since e is θ-SN, $\theta\langle \text{e}\rangle$ is SN. Therefore the entire term is SN. □

Lemma B.4.

1. $\forall \text{R}, \tau_1, \tau_2, \text{c}_1, \text{c}_2$, *if* c_1 *and* c_2 *are θ2-SN, then* $\text{reify}_* \text{ R } \tau_1 \ \tau_2 \text{ c}_1 \text{ c}_2$ *is θ2-SN.*
2. $\forall \text{R}, \tau_1, \tau_2, \text{c}_1, \text{c}_2$, *if* c_1 *and* c_2 *are θ2-SN, then* $\text{reflect}_* \text{ R } \tau_1 \ \tau_2 \text{ c}_1 \text{ c}_2$ *is θ2-SN.*
3. $\forall \text{R}, \kappa, \tau, \text{c}$, *if* c α *is θ2-SN for all α, then* $\text{reify}_\square \text{ R } \tau \text{ c}$ *is θ2-SN.*
4. $\forall \text{R}, \kappa, \tau, \text{c}$, *if* c α *is θ2-SN for all α, then* $\text{reflect}_\square \text{ R } \tau \text{ c}$ *is θ2-SN.*
5. $\forall \text{R}, \tau, \text{c}$, *if* c χ *is θ2-SN for all χ, then* $\text{reify}_\Delta \text{ R } \tau \text{ c}$ *is θ2-SN.*

6. $\forall \mathrm{R}, \tau, \mathrm{c}$, *if* $\mathrm{c}\ \chi$ *is* θ*2-SN for all* χ*, then* $\mathrm{reflect}_{\Delta}\ \mathrm{R}\ \tau\ \mathrm{c}$ *is* θ*2-SN.*

Proof. Each case is similar. (3) is representative.

Let c be such that $\mathrm{c}\ \alpha$ is θ2-SN for all α, and let e be θ-SN. We must show that $\theta\langle \mathrm{reify}_{\square}\ \mathrm{R}\ \kappa\ \mathrm{c}\ \mathrm{e}\rangle$ is SN. By Lemma A.8, we have that

$$\begin{aligned} & \theta\langle \mathrm{reify}_{\square}\ \mathrm{R}\ \kappa\ \tau\ \mathrm{c}\ \mathrm{e}\rangle \\ \equiv_{\beta}\ & \theta\langle \mathrm{coerce}_{\square}\ \mathrm{R}\ \kappa\ (\lambda\alpha : \kappa.\ \mathrm{Var}\ (\mathrm{UId}\ (\tau\ \alpha)))\ \tau\ \mathrm{c} \\ & (\mathrm{dist}_{\square}\ \mathrm{R}\ \kappa\ \tau\ \mathrm{e})\rangle \end{aligned}$$

By Lemma B.3, $\mathrm{coerce}_{\square}\ \mathrm{R}\ \kappa\ (\lambda\alpha : \kappa.\ \mathrm{Var}\ (\mathrm{UId}\ (\tau\ \alpha)))\ \tau\ \mathrm{c}$ is θ2-SN. Also by Lemma B.3, $\mathrm{dist}_{\square}\ \mathrm{R}\ \kappa\ \tau$ is θ2-SN. Therefore, $(\mathrm{dist}_{\square}\ \mathrm{R}\ \kappa\ \tau\ te)$ is θ-SN, which in turn gives that the entire term $\theta\langle \mathrm{coerce}_{\square}\ \ldots\rangle$ is SN. Therefore, $\mathrm{reify}_{\square}\ \mathrm{R}\ \kappa\ \tau\ \mathrm{c}$ is θ2-SN. □

Lemma B.5. *If* $\tau_1 \rightsquigarrow \tau_2 = \mathrm{c}$*, then* c *is* θ*2-SN.*

Proof. Straightforward by induction on the height of the derivation of $\tau_1 \rightsquigarrow \tau_2 = \mathrm{c}$, and by Lemmas B.4 and B.3. □

Lemma B.6.

1. $\forall \mathrm{R}, \tau, \mathrm{x}$: *If* x *is a variable, then* $\mathrm{mkVar}\ \mathrm{R}\ \tau\ \mathrm{x}$ *is* θ*-SN.*
2. $\forall \mathrm{R}, \tau_1, \tau_2, \mathrm{q}$: *If* q x *is* θ*-SN for any variable* x*, then* $\mathrm{mkAbs}_{*}\ \mathrm{R}\ \tau_1\ \tau_2\ \mathrm{q}$ *is* θ*-SN.*
3. $\forall \mathrm{R}, \tau_1, \tau_2, \mathrm{q}_1, \mathrm{q}_2$: *If* q_1 *and* q_2 *are* θ*-SN, then* $\mathrm{mkApp}_{*}\ \mathrm{R}\ \tau_1\ \tau_2\ \mathrm{q}_1\ \mathrm{q}_2$ *is* θ*-SN.*
4. $\forall \mathrm{R}, \kappa, \tau, \mathrm{q}$: *If* q α *is* θ*-SN for any variable* α*, then* $\mathrm{mkAbs}_{\square}\ \mathrm{R}\ \kappa\ \tau\ \mathrm{q}$ *is* θ*-SN.*
5. $\forall \mathrm{R}, \kappa, \tau, \mathrm{q}, \tau_1$: *If* q *is* θ*-SN, then* $\mathrm{mkApp}_{\square}\ \mathrm{R}\ \kappa\ \tau\ \mathrm{q}\ \tau_1$ *is* θ*-SN.*
6. $\forall \mathrm{R}, \tau, \mathrm{q}$: *If* q χ *is* θ*-SN for any variable* χ*, then* $\mathrm{mkAbs}_{\Delta}\ \mathrm{R}\ \tau\ \mathrm{q}$ *is* θ*-SN.*
7. $\forall \mathrm{R}, \tau, \mathrm{q}, \kappa$: *If* q *is* θ*-SN, then* $\mathrm{mkApp}_{\Delta}\ \mathrm{R}\ \kappa\ \tau\ \mathrm{q}\ \kappa$ *is* θ*-SN.*

Proof. Each case is similar. (3) is representative.

Suppose q_1 and q_2 are θ-SN. By Lemma A.1, $\theta\langle \mathrm{mkApp}_{*}\ \mathrm{R}\ \tau_1\ \tau_2\ \mathrm{q}_1\ \mathrm{q}_2\rangle \equiv_{\beta} \mathrm{f}_2\ \tau_1\ \tau_2\ \theta\langle \mathrm{q}_1\rangle\ \theta\langle \mathrm{q}_2\rangle$. Since f_2 is a variable, τ_1 and τ_2 are types, and $\theta\langle \mathrm{q}_1\rangle$ and $\theta\langle \mathrm{q}_2\rangle$ are SN, the entire term is SN as required. □

Lemma B.7. *If* $\Gamma \vdash \mathrm{e} : \tau \blacktriangleright \mathrm{q}$*, then* q *is* θ*-SN.*

Proof. Straightforward by induction on the height of the derivation $\Gamma \vdash \mathrm{e} : \tau \blacktriangleright \mathrm{q}$, and by Lemma B.6 and Lemma B.5. □

Lemma B.8. *If* $\Gamma \vdash \mathrm{e} : \tau \blacktriangleright \mathrm{q}$*, then* q *is SN.*

Proof. Follows from Lemma B.7 and Lemma B.1. □

C. Proofs

Definition C.1. *A PTS is called functional (or singly-sorted) if*

1. $(c : s_1), (c : s_2) \in \mathcal{A} \ \Rightarrow\ s_1 = s_2$
2. $(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R} \ \Rightarrow\ s_3 = s_3'$

Theorem C.1 (Uniqueness of types in a functional PTS [5])**.** *Let* λS *be a singly-sorted PTS. Then*

$$\Gamma \vdash \mathrm{A} : \mathrm{B}_1\ \&\ \Gamma \vdash \mathrm{A} : \mathrm{B}_2 \ \Rightarrow\ \mathrm{B}_1 \equiv_{\beta} \mathrm{B}_2$$

Definition C.2. *A PTS is called injective if*

1. *It is functional*
2. $(s_1 : s_2), (s_1' : s_2) \in \mathcal{A} \ \Rightarrow\ s_1 = s_1'$
3. $(s_1, s_2, s_3), (s_1, s_2', s_3) \in \mathcal{R} \ \Rightarrow\ s_2 = s_2'$

Theorem C.2. *Typechecking of an injective PTS is decidable [6].*

Lemma C.1. *System U is functional.*

Proof.

1. Each constant c is in the left-hand position of at most one axiom.
2. Suppose $(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R}$. Then $s_2 = s_3$ and $s_2 = s_3'$. Therefore, $s_3 = s_3'$.

□

Lemma C.2. *System U is injective.*

Proof.

1. By Lemma C.1.
2. Each sort s is in the right-hand position of at most one axiom.
3. Suppose $(s_1, s_2, s_3), (s_1, s_2', s_3) \in \mathcal{R}$. Then $s_2 = s_3$ and $s_2' = s_3$. Therefore, $s_2 = s_2'$.

□

Theorem C.3. *Type checking is decidable for* λU.

Proof. Follows from Lemma C.2 and Theorem C.2. □

Lemma C.3. *If* $\Gamma \vdash \tau : \kappa : \square$*, then* τ *has a normal form.*

Proof. The proof is by an embedding the types of System λU into the terms of System F, and the kinds of System λU into the types of System F. □

Theorem 3.2. *If* $\Gamma \vdash \tau : *$*, and* τ *is a normal form, then* τ *is of one of the following forms:*

$\alpha\ A_1\ \ldots\ A_n$	where α is a type variable.
$\tau_1 \to \tau_2$	where $\Gamma \vdash \tau_1 : *$
$\Pi\alpha : \kappa.\ \tau_1$	where $\Gamma \vdash \kappa : \square$
$\Pi\chi : \square.\ \tau_1$	

Proof. Straightfoward by Lemma C.3 and then by induction on the height of $\Gamma \vdash \tau : *$. □

Theorem 3.3 (Decomposition of product types)**.** *For any legal* $(*, *)$ *product* $\tau_1 \to \tau_2$*, any legal* $(\square, *)$ *product* $\Pi\alpha : \kappa.\ \tau$*, and any legal* $(\Delta, *)$ *product* $\Pi\chi : \square.\ \tau$*, we have:*

$$\begin{aligned} \tau_1 \to \tau_2 &\equiv_{\beta} \pi_{*}\ \tau_1\ \tau_2 \\ \Pi\alpha : \kappa.\ \tau &\equiv_{\beta} \pi_{\square}\ \kappa\ (\lambda\alpha : \kappa.\ \tau) \\ \Pi\chi : \square.\ \tau &\equiv_{\beta} \pi_{\Delta}\ (\lambda\chi : \square.\ \tau) \end{aligned}$$

Proof. Straightforward by the definitions of π_{*}, $\pi_{\square}$, and π_{Δ}.

$$\begin{aligned} & \pi_{*}\ \tau_1\ \tau_2 \\ =\ & (\lambda\alpha : *.\ \lambda\beta : *.\ \alpha \to \beta)\ \tau_1\ \tau_2 \\ \equiv_{\beta}\ & \tau_1 \to \tau_2 \end{aligned}$$

$$\begin{aligned} & \pi_{\square}\ \kappa\ (\lambda\alpha : \kappa.\ \tau) \\ =\ & (\lambda\chi : \square.\ \lambda\alpha : \chi \to *.\Pi\beta : \chi.\ \alpha\ \beta)\ \kappa\ (\lambda\alpha : \kappa.\ \tau) \\ \equiv_{\beta}\ & (\lambda\alpha : \kappa \to *.\Pi\beta : \kappa.\ \alpha\ \beta)\ (\lambda\alpha : \kappa.\ \tau) \\ \equiv_{\beta}\ & \Pi\beta : \kappa.\ (\lambda\alpha : \kappa.\ \tau)\ \beta \\ \equiv_{\alpha}\ & \Pi\alpha : \kappa.\ (\lambda\alpha : \kappa.\ \tau)\ \alpha \\ \equiv_{\beta}\ & \Pi\alpha : \kappa.\ \tau \end{aligned}$$

$$\begin{aligned} & \pi_{\Delta}\ (\lambda\chi : \square.\ \tau) \\ =\ & (\lambda\alpha : \square \to *.\Pi\chi : \square.\ \alpha\ \chi)\ (\lambda\chi : \square.\ \tau) \\ \equiv_{\beta}\ & \Pi\chi : \square.\ (\lambda\chi : \square.\ \tau)\ \chi \\ \equiv_{\beta}\ & \Pi\chi : \square.\ \tau \end{aligned}$$

□

Theorem 4.1 (Kinds of type representations)**.** *If* $\Gamma \vdash \tau : *$ *and* $\Gamma \vdash \tau : * \rhd \sigma$*, then* $\Gamma \vdash \sigma$: U.

Proof. Straightforward by induction on the height of the derivation $\Gamma \vdash \tau : * \rhd \sigma$, and by the types of the constructors in Definition 4.2. □

Theorem 4.2. If $\Gamma \vdash \tau : *$, then UId $\overline{\tau} \equiv_\beta \tau$.

Proof. By induction the size of the type τ.

Suppose τ is of the form $\alpha\ \tau_1\ \ldots\ \tau_n$. By Theorem A.4, UId $\overline{\tau} \equiv_\beta (\lambda\alpha : *.\ \alpha)\ \tau \equiv_\beta \tau$.

Suppose τ is of the form $\tau_1 \to \tau_2$ and $\Gamma \vdash \tau_1 : *$. By the induction hypothesis, UId $\overline{\tau_1} \equiv_\beta \tau_1$ and UId $\overline{\tau_2} \equiv_\beta \tau_2$. By Theorem 3.3, $\pi_*\ \tau_1\ \tau_2 \equiv_\beta \tau$. Therefore, by Theorem A.4, UId $\overline{\tau} \equiv_\beta \pi_*$ (UId $\overline{\tau_1}$) (UId $\overline{\tau_2}$) $\equiv_\beta \pi_*\ \tau_1\ \tau_2 \equiv_\beta \tau$.

Suppose τ is of the form $\Pi\alpha : \kappa.\ \tau_1$ and $\Gamma \vdash \kappa : \Box$. By the induction hypothesis, UId $\overline{\tau_1} \equiv_\beta \tau_1$. By Theorem 3.3, $\pi_\Box\ \kappa\ \tau_1 \equiv_\beta \tau$. Therefore, by Theorem A.4, UId $\overline{\tau} \equiv_\beta \pi_\Box\ \kappa$ (UId $\overline{\tau_1}$) $\equiv_\beta \pi_\Box\ \kappa\ \tau_1 \equiv_\beta \tau$.

Suppose τ is of the form $\Pi\chi : \Box.\ \tau_1$ and $\Gamma \vdash \kappa : \Box$. By the induction hypothesis, UId $\overline{\tau_1} \equiv_\beta \tau_1$. By Theorem 3.3, $\pi_\Delta\ \tau_1 \equiv_\beta \tau$. Therefore, by Theorem A.4, UId $\overline{\tau} \equiv_\beta \pi_\Delta$ (UId $\overline{\tau_1}$) $\equiv_\beta \pi_\Delta\ \tau_1 \equiv_\beta \tau$. □

Lemma C.4 (Semantics of U Constructors)**.** *Let* F = Fold[F_1, F_2, F_3, F_4]. *Then,*

$$\begin{aligned}
&F\ (\text{Prod}_*\ \tau_1\ \tau_2) &&\equiv_\beta F_2\ (F\ \tau_1)(F\ \tau_2)\\
&F\ (\text{Prod}_\Box\ \kappa\ \tau) &&\equiv_\beta F_3\ \kappa\ (\lambda\beta : \kappa.\ F\ (\tau\ \beta))\\
&F\ (\text{Prod}_\Delta\ \tau) &&\equiv_\beta F_4\ (\lambda\chi : \Box.\ F(\tau\ \chi))
\end{aligned}$$

Proof. By the definition of Fold, we have that

$$F = \lambda\alpha : U.\ \alpha\ F_1\ F_2\ F_3\ F_4$$

$$\begin{aligned}
&F\ (\text{Prod}_*\ \tau_1\ \tau_2)\\
\equiv_\beta\ &(\text{Prod}_*\ \tau_1\ \tau_2)\ F_1\ F_2\ F_3\ F_4\\
\equiv_\beta\ &F_2\ (\tau_1\ F_1\ F_2\ F_3\ F_4)\ (\tau_2\ F_1\ F_2\ F_3\ F_4)\\
\equiv_\beta\ &F_2\ (F\ \tau_1)\ (F\ \tau_2)
\end{aligned}$$

$$\begin{aligned}
&F\ (\text{Prod}_\Box\ \kappa\ \tau)\\
\equiv_\beta\ &(\text{Prod}_\Box\ \kappa\ \tau)\ F_1\ F_2\ F_3\ F_4\\
\equiv_\beta\ &F_3\ \kappa\ (\lambda\beta : \kappa.\ \tau\ \alpha\ F_1\ F_2\ F_3\ F_4)\\
\equiv_\beta\ &F_3\ \kappa\ (\lambda\beta : \kappa.\ F\ (\tau\ \beta))
\end{aligned}$$

$$\begin{aligned}
&F\ (\text{Prod}_\Delta\ \tau)\\
\equiv_\beta\ &(\text{Prod}_\Delta\ \tau)\ F_1\ F_2\ F_3\ F_4\\
\equiv_\beta\ &F_4\ (\lambda\chi : \Box.\ \tau\ \chi\ F_1\ F_2\ F_3\ F_4)\\
\equiv_\beta\ &F_4\ (\lambda\chi : \Box.\ F\ (\tau\ \chi))
\end{aligned}$$

□

Lemma C.5 (Kind of type representations in representation environment)**.** *If* $\Gamma \vdash \tau : *$*, then* $\overline{\Gamma} \vdash \overline{\tau}$: U.

Proof. Sketch: By theorem 4.1, we have that $\Gamma \vdash \overline{\tau}$: U. Since Γ and $\overline{\Gamma}$ are equivalent with respect to type and kind bindings, $\overline{\Gamma} \vdash \overline{\tau}$: U. □

Lemma C.6. *If* τ *is a normal form and* $\Gamma, \chi : \Box \vdash \tau : *$ *and* $\Gamma \vdash \kappa : \Box$*, then* $\overline{\tau}[\chi := \kappa] \equiv \overline{\tau[\chi := \kappa]}$.

Proof. Straightforward by Lemma 3.2, and by induction on the height of $\Gamma, \chi : \Box \vdash \tau : *$. □

Lemma C.7. *If* $\Gamma \vdash e : \tau \blacktriangleright q$*, then* R : U $\to *, \overline{\Gamma} \vdash q$: PExp R $\overline{\tau}$.

```
Witness[R, d*, f*, c*, d□, f□, c□, d△, f△, c△] =
  λα : *.
  λ f :
    (Dist* R → Dist□ R → Dist△ R →
     Factor* R → Factor□ R → Factor△ R →
     Coerce* R → Coerce□ R → Coerce△ R →
     α).
  f d* f* c* d□ f□ c□ d△ f△ c△
```

Figure 20. Witness Context

Proof. Straightforward by induction on the height of the derivation of $\Gamma \vdash e : \tau \blacktriangleright q$, and Lemmas C.5 and C.6 and Theorems A.9 and A.10. □

Theorem 5.2 (Types of quotations)**.** *If* $\langle\rangle \vdash e : \tau : *$*, and* quote(e) = q*, then* $\langle\rangle \vdash$ q : Exp $\overline{\tau}$.

Proof. By definition of quote(·), we have that and $q = \lambda$R : U $\to$ $*.q_1$ and $\langle\rangle \vdash e : \tau \blacktriangleright q_1$. By Lemma C.7, R : U $\to *, \overline{\langle\rangle} \vdash q_1$: PExp R $\overline{\tau}$. Therefore, $\overline{\langle\rangle} \vdash q$: (ΠR : U $\to *$. PExp R $\overline{\tau}$). But $\overline{\langle\rangle} = \langle\rangle$, and (ΠR : U $\to *$. PExp R $\overline{\tau}$) $\equiv_\beta$ Exp $\overline{\tau}$, so $\langle\rangle \vdash q$: Exp $\overline{\tau}$ as required. □

Theorem 5.3. *If* quote(e) = q*, then* q *is strongly normalizing.*

Proof. Suppose quote(e) = q. Then q = λR : U $\to *$. q_1 and $\langle\rangle \vdash$ e : $\tau \blacktriangleright q_1$, for some τ and q_1. By Lemma B.8, q_1 is strongly normalizing. Therefore, q is strongly normalizing.

□

Definition C.3 (Witness Context)**.** *For R,*d_**,*f_**,*c_**,* $d_\Box$*,* $f_\Box$*,* $c_\Box$*,* d_Δ*,* f_Δ*,*c_Δ *such that:*

- $\langle\rangle \vdash$ R : U $\to *$
- $\langle\rangle \vdash$ d_* : Dist_*
- $\langle\rangle \vdash$ f_* : Factor_*
- $\langle\rangle \vdash$ c_* : Coerce_*
- $\langle\rangle \vdash$ $d_\Box$: $\text{Dist}_\Box$
- $\langle\rangle \vdash$ $f_\Box$: $\text{Factor}_\Box$
- $\langle\rangle \vdash$ $c_\Box$: $\text{Coerce}_\Box$
- $\langle\rangle \vdash$ d_Δ : Dist_Δ
- $\langle\rangle \vdash$ f_Δ : Factor_Δ
- $\langle\rangle \vdash$ c_Δ : Coerce_Δ

We define the term Witness[R,d_**,*f_**,*c_**,* $d_\Box$*,* $f_\Box$*,* $c_\Box$*,* d_Δ*,* f_Δ*,*c_Δ*] as in Figure 20.*

Theorem C.4 (Type of witnesses)**.** *Let R,* d_**,*f_**,*c_**,* $d_\Box$*,* $f_\Box$*,* $c_\Box$*,* d_Δ*,* f_Δ*,*c_Δ *be as in Definition C.3. Then,*
$\langle\rangle \vdash$ Witness[R, d_*, f_*, c_*, $d_\Box$, $f_\Box$, $c_\Box$, d_Δ, f_Δ, c_Δ] : Witness R

Definition C.4 (Identity Witness Function)**.** *Let* f *be a term and* R : U $\to *$ *be a type. Then f is an identity witness function if one of the following is true:*

- $\langle\rangle \vdash$ f : Dist_* R *and for all* τ_1*,* τ_2*, and e, we have that* f τ_1 τ_2e $\equiv_\beta$ e.
- $\langle\rangle \vdash$ f : Factor_* R *and for all* τ_1*,* τ_2*, and e, we have that* f τ_1 τ_2e $\equiv_\beta$ e.
- $\langle\rangle \vdash$ f : Coerce_* R *and for all* τ_1*,* τ_2*,* τ_3*,* τ_4*,* c_1*,* c_2*, and* e *such that* c_1 e_1 $\equiv_\beta$ e_1 *and* c_2 e_2 $\equiv_\beta$ e_2 *for all* e_1*,* e_2*, we have that* f τ_1 τ_2 τ_3 τ_4 c_1 c_2 e $\equiv_\beta$ e.
- $\langle\rangle \vdash$ f : $\text{Dist}_\Box$ R *and for all* κ*,* τ*, and* e *we have that* f κ τ e $\equiv_\beta$ e.

- $\langle\rangle \vdash \mathrm{f} : \mathrm{Factor}_\square\ \mathrm{R}$ *and for all* κ, τ, *and* e *we have that* $\mathrm{f}\ \kappa\ \tau\ \mathrm{e} \equiv_\beta \mathrm{e}$.
- $\langle\rangle \vdash \mathrm{f} : \mathrm{Coerce}_\square\ \mathrm{R}$ *and for all* κ, τ_1, τ_2, c_1 *and* e *such that* $\mathrm{c}_1\ \tau_3\ \mathrm{e}_1 \equiv_\beta \mathrm{e}_1$ *for all* τ_3 *and* e_1, *we have that* $\mathrm{f}\ \kappa\ \tau_1\ \tau_2\ \mathrm{c}_1\ \mathrm{e} \equiv_\beta \mathrm{e}$.
- $\langle\rangle \vdash \mathrm{f} : \mathrm{Dist}_\Delta\ \mathrm{R}$ *and for all* τ, *and* e *we have that* $\mathrm{f}\ \tau\ \mathrm{e} \equiv_\beta \mathrm{e}$.
- $\langle\rangle \vdash \mathrm{f} : \mathrm{Factor}_\Delta\ \mathrm{R}$ *and for all* τ, *and* e *we have that* $\mathrm{f}\ \tau\ \mathrm{e} \equiv_\beta \mathrm{e}$.
- $\langle\rangle \vdash \mathrm{f} : \mathrm{Coerce}_\Delta\ \mathrm{R}$ *and for all* τ_1, τ_2, c_1 *and* e *such that* $\mathrm{c}_1\ \kappa\ \mathrm{e}_1 \equiv_\beta \mathrm{e}_1$ *for all* κ *and* e_1, *we have that* $\mathrm{f}\ \tau_1\ \tau_2\ \mathrm{c}_1\ \mathrm{e} \equiv_\beta \mathrm{e}$.

Definition C.5 (Identity Witness)**.** *Let R,* d_*,f_*,c_*, $d_\square$, $f_\square$, $c_\square$, d_Δ, f_Δ,c_Δ *be as in Definition C.3. Then Witness[R,*d_*,f_*,c_*, $d_\square$, $f_\square$, $c_\square$, d_Δ, f_Δ,c_Δ*] is an identity witness if* d_*,f_*,c_*, $d_\square$, $f_\square$, $c_\square$, d_Δ, f_Δ, *and* c_Δ *are all identity witness functions.*

Lemma C.8 (Semantics of Identity Witnesses)**.** *Let* w *be an identity witness. Then:*

1)	$\mathrm{w}\ (\mathrm{Dist}_*\ \mathrm{R})\ \mathrm{w}_1\ \tau_1\ \tau_2\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
2)	$\mathrm{w}\ (\mathrm{Factor}_*\ \mathrm{R})\ \mathrm{w}_2\ \tau_1\ \tau_2\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
3)	If $\mathrm{c}_1\ \mathrm{e}_1 \equiv_\beta \mathrm{e}_1$for all e_1, and $\mathrm{c}_2\ \mathrm{e}_2 \equiv_\beta \mathrm{e}_2$for all e_2, then $\mathrm{w}\ (\mathrm{Coerce}_*\ \mathrm{R})\ \mathrm{w}_3\ \tau_1\ \tau_2\ \sigma_1\ \sigma_2\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
4)	$\mathrm{w}\ (\mathrm{Dist}_\square\ \mathrm{R})\ \mathrm{w}_4\ \kappa\ \tau\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
5)	$\mathrm{w}\ (\mathrm{Factor}_\square\ \mathrm{R})\ \mathrm{w}_5\ \kappa\ \tau\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
6)	If $\mathrm{c}\ \mathrm{e}_1 \equiv_\beta \mathrm{e}_1$ for all e_1, then $\mathrm{w}\ (\mathrm{Coerce}_\square\ \mathrm{R})\ \mathrm{w}_6\ \kappa\ \tau\ \sigma\ \mathrm{c}\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
7)	$\mathrm{w}\ (\mathrm{Dist}_\Delta\ \mathrm{R})\ \mathrm{w}_7\ \kappa\ \tau\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
8)	$\mathrm{w}\ (\mathrm{Factor}_\Delta\ \mathrm{R})\ \mathrm{w}_8\ \tau\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$
9)	If $\mathrm{c}\ \mathrm{e}_1 \equiv_\beta \mathrm{e}_1$ for all e_1, then $\mathrm{w}\ (\mathrm{Coerce}_\Delta\ \mathrm{R})\ \mathrm{w}_9\ \tau\ \sigma\ \mathrm{c}\ \mathrm{e}$	$\equiv_\beta \mathrm{e}$

Proof. Straightforward, by definitions A.4, C.4 and C.5, and a few steps of β-reduction. □

Definition C.6. *For any identity witness* w*, and any terms* $\mathrm{f}_1, \ldots, \mathrm{f}_6$ *(where* $\mathrm{w}, \mathrm{f}_1, \ldots, \mathrm{f}_6$ *may be inferred by context),* $\phi\langle\mathrm{e}\rangle$ *denotes the term* $\mathrm{e}\ \mathrm{w}\ \mathrm{f}_1\ \ldots\ \mathrm{f}_6$.

Definition C.7. *Let* f *be a term. If* $\phi\langle\mathrm{f\ a}\rangle \equiv_\beta \phi\langle\mathrm{a}\rangle$ *for any term* a*, we say that* f *is a* ϕ*-identity function, or that* f *is* ϕ*-id.*

Lemma C.9. *1. For all* R, τ_1, τ_2*:* $\mathrm{dist}_*\ \mathrm{R}\ \tau_1\ \tau_2$ *is* ϕ*-id.*
2. *For all* R, τ_1, τ_2*:* $\mathrm{factor}_*\ \mathrm{R}\ \tau_1\ \tau_2$ *is* ϕ*-id.*
3. *For all* R, τ_1, τ_2, σ_1, σ_2, c_1, c_2*: If* c_1 *and* c_2 *are* ϕ*-id, then* $\mathrm{coerce}_*\ \mathrm{R}\ \tau_1\ \tau_2\ \sigma_1\ \sigma_2\ \mathrm{c}_1\ \mathrm{c}_2$ *is* ϕ*-id.*
4. *For all* R, κ, τ*:* $\mathrm{dist}_\square\ \mathrm{R}\ \kappa\ \tau$ *is* ϕ*-id.*
5. *For all* R, κ, τ*:* $\mathrm{factor}_\square\ \mathrm{R}\ \kappa\ \tau$ *is* ϕ*-id.*
6. *For all* R, κ, τ, σ, c*: If* c *is* ϕ*-id, then* $\mathrm{coerce}_\square\ \mathrm{R}\ \kappa\ \tau\ \sigma\ \mathrm{c}$ *is* ϕ*-id.*
7. *For all* R, τ*:* $\mathrm{dist}_\Delta\ \mathrm{R}\ \tau$ *is* ϕ*-id.*
8. *For all* R, τ*:* $\mathrm{factor}_\Delta\ \mathrm{R}\ \tau$ *is* ϕ*-id.*
9. *For all* R, τ, σ, c*: If* c *is* ϕ*-id, then* $\mathrm{coerce}_\Delta\ \mathrm{R}\ \tau\ \sigma\ \mathrm{c}$ *is* ϕ*-id.*

Proof. Straightforward, by Lemmas A.7 and C.8. □

Lemma C.10. *1. For all* R, τ_1, τ_2, c_1, c_2*: If* c_1 *and* c_2 *are* ϕ*-id, then* $\mathrm{reify}_*\ \mathrm{R}\ \tau_1\ \tau_2\ \mathrm{c}_1\ \mathrm{c}_2$ *is* ϕ*-id.*
2. *For all* R, τ_1, τ_2, c_1, c_2*: If* c_1 *and* c_2 *are* ϕ*-id, then* $\mathrm{reflect}_*\ \mathrm{R}\ \tau_1\ \tau_2\ \mathrm{c}_1\ \mathrm{c}_2$ *is* ϕ*-id.*
3. *For all* R, κ, τ, c*: If* c α *is* ϕ*-id for any* α*, then* $\mathrm{reify}_\square\ \mathrm{R}\ \kappa\ \tau\ \mathrm{c}$ *is* ϕ*-id.*
4. *For all* R, κ, τ, c*: If* c α *is* ϕ*-id for any* α*, then* $\mathrm{reflect}_\square\ \mathrm{R}\ \kappa\ \tau\ \mathrm{c}$ *is* ϕ*-id.*
5. *For all* R, τ, c*: If* c χ *is* ϕ*-id for any* χ*, then* $\mathrm{reify}_\Delta\ \mathrm{R}\ \tau\ \mathrm{c}$ *is* ϕ*-id.*
6. *For all* R, τ, c*: If* c χ *is* ϕ*-id for any* χ*, then* $\mathrm{reflect}_\Delta\ \mathrm{R}\ \tau\ \mathrm{c}$ *is* ϕ*-id.*

Proof. Straightforward by Lemma A.8 and Lemma C.9. □

Lemma C.11 (Coercions based on identity witnesses)**.** *If* $\tau_1 \rightsquigarrow \tau_2 = \mathrm{c}$*, then* c *is* ϕ*-id.*

Proof. Straightforward by induction on the height of $\tau_1 \rightsquigarrow \tau_2 = \mathrm{c}$, and by Lemma C.9 and Lemma C.10. □

Lemma C.12. *Suppose* F, w, abs_*, app_*, $\mathrm{abs}_\square$, $\mathrm{app}_\square$, abs_Δ, *and* app_Δ *are as in Definition A.2, and that* w *is an identity witness. Suppose* $\mathrm{f} = \mathit{fold}$*[F, w,* abs_*, app_*, $\mathit{abs}_\square$, $\mathit{app}_\square$, abs_Δ, app_Δ*]. Then for any context* Γ*, term* e*, and types* τ *and* τ_1 *such that* $\Gamma \vdash \mathrm{e} : \Pi\alpha : \kappa.\ \tau$ *and* $\Gamma \vdash \tau_1 : \kappa$*, we have that:*

$$\mathrm{f}\ \overline{\tau[\alpha := \tau_1]}\ \overline{\mathrm{e}\ \tau_1} \equiv_\beta \mathrm{app}_\square\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau})\ (\mathrm{f}\ \overline{\Pi\alpha : \kappa.\ \tau}\ \overline{\mathrm{e}})\ \tau_1$$

Proof. Be the definition of fold[...] and $\overline{\mathrm{e}}$, we have that:

$$\begin{aligned} &\mathrm{f}\ \overline{\tau[\alpha := \tau_1]}\ \overline{\mathrm{e}\ \tau_1} \\ \equiv_\beta\ &(\mathrm{c}\ (\mathrm{mkApp}_\square\ \mathrm{R}\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau})\ \overline{\mathrm{e}}\ \tau_1)) \\ &\quad \mathrm{w}\ \mathrm{abs}_*\ \mathrm{app}_*\ \mathrm{abs}_\square\ \mathrm{app}_\square\ \mathrm{abs}_\Delta\ \mathrm{app}_\Delta \end{aligned}$$

where $\overline{\tau}[\alpha := \tau_1] \rightsquigarrow \overline{\tau[\alpha := \tau_1]} = \mathrm{c}$. By Lemma C.11, we have that:

$$\begin{aligned} &(\mathrm{c}\ (\mathrm{mkApp}_\square\ \mathrm{R}\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau})\ \overline{\mathrm{e}}\ \tau_1)) \\ &\quad \mathrm{w}\ \mathrm{abs}_*\ \mathrm{app}_*\ \mathrm{abs}_\square\ \mathrm{app}_\square\ \mathrm{abs}_\Delta\ \mathrm{app}_\Delta \\ \equiv_\beta\ &(\mathrm{mkApp}_\square\ \mathrm{R}\ \kappa\ (\lambda\alpha : \kappa.\ \overline{\tau})\ \overline{\mathrm{e}}\ \tau_1) \\ &\quad \mathrm{w}\ \mathrm{abs}_*\ \mathrm{app}_*\ \mathrm{abs}_\square\ \mathrm{app}_\square\ \mathrm{abs}_\Delta\ \mathrm{app}_\Delta \end{aligned}$$

The result follows from a few steps of straightforward β-reduction. □

Theorem 5.1. *If* $\Gamma \vdash \mathrm{e} : \tau$ *and* $\langle\rangle \vdash \mathrm{F} : \mathrm{U} \to *$*, then* $\overline{\Gamma} \vdash \overline{\mathrm{e}} : \mathrm{PExp\ F}\ \overline{\tau}$.

Proof. We have that $\overline{\mathrm{e}} = q[\mathrm{R} := \mathrm{F}]$ and $\Gamma \vdash \mathrm{e} : \tau \blacktriangleright q$. By Lemma C.7, we have that $\mathrm{R} : \mathrm{U} \to *, \overline{\Gamma} \vdash q : \mathrm{PExp\ R}\ \overline{\tau}$. By weakening, $\overline{\Gamma} \vdash \mathrm{F} : \mathrm{U} \to *$. Therefore, $\overline{\Gamma} \vdash \overline{\mathrm{e}} : \mathrm{PExp\ F}\ \overline{\tau}$ as required. □

C.1 unquote

Our self-recognizer unquote is defined in `lib/unquotelib.pts`. unquote= $\lambda\alpha$: U. λe : Exp α. fold[UId, witnessUId, id_*, id_*, $\mathrm{id}_\square$, $\mathrm{id}_\square$, id_Δ, id_Δ] α (e UId).

Lemma C.13 (Types of unquote case functions)**.**

$\langle\rangle \vdash \mathrm{id}_* : (\Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}.\ (\mathrm{UId}\ \alpha \to \mathrm{UId}\ \beta) \to \mathrm{UId}\ (\mathrm{Prod}_*\ \alpha\ \beta))$
$\langle\rangle \vdash \mathrm{id}_* : (\Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}.\ \mathrm{UId}\ (\mathrm{Prod}_*\ \alpha\ \beta) \to \mathrm{UId}\ \alpha \to \mathrm{UId}\ \beta)$
$\langle\rangle \vdash \mathrm{id}_\square : (\Pi\chi : \square.\ \Pi\alpha : \chi \to \mathrm{U}.$
$\quad (\Pi\beta : \chi.\ \mathrm{UId}\ (\alpha\ \beta)) \to \mathrm{UId}\ (\mathrm{Prod}_\square\ \chi\ \alpha))$
$\langle\rangle \vdash \mathrm{id}_\square : (\Pi\chi : \square.\ \Pi\alpha : \chi \to \mathrm{U}.$
$\quad \mathrm{UId}\ (\mathrm{Prod}_\square\ \chi\ \alpha) \to (\Pi\beta : \chi.\ \mathrm{UId}\ (\alpha\ \beta)))$
$\langle\rangle \vdash \mathrm{id}_\Delta : (\Pi\alpha : \square \to \mathrm{U}.\ (\Pi\chi : \square.\ \mathrm{UId}\ \alpha\ \chi) \to \mathrm{UId}\ (\mathrm{Prod}_\Delta\ \alpha))$
$\langle\rangle \vdash \mathrm{id}_\Delta : (\Pi\alpha : \square \to \mathrm{U}.\ \mathrm{UId}\ (\mathrm{Prod}_\Delta\ \alpha) \to (\Pi\chi : \square.\ \mathrm{UId}\ \alpha\ \chi))$

Proof. Machine checked. □

Theorem 6.1 (Type of unquote)**.**

$$\langle\rangle \vdash \mathrm{unquote} : (\Pi\alpha : \mathrm{U}.\ \mathrm{Exp}\ \alpha \to \mathrm{UId}\ \alpha)$$

Proof. Follows directly from Theorem A.8. □

Lemma C.14. witnessUId *is an identity witness.*

Proof. Straightforward. Each witness function is an identity witness function. □

Lemma C.15. *Let* f = fold[UId, witnessUId, id_*, id_*, $\mathrm{id}_\Box$, $\mathrm{id}_\Box$, id_Δ, id_Δ]. *If* $\Gamma \vdash \mathrm{e} : \tau \blacktriangleright \mathrm{q}$, *then* $\mathrm{f}\ \overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{e}$.

Proof. Straightforward by induction on the height of $\Gamma \vdash \mathrm{e} : \tau \blacktriangleright \mathrm{q}$, and by Lemmas C.14 and C.12 and Theorem A.11. Note that each fold function is an identity function. □

Theorem 6.2 (Correctness of unquote)**.**
If $\langle\rangle \vdash \mathrm{e} : \tau$ *and* quote(e) = q, *then* unquote $\overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{e}$.

Proof. Let f = fold[UId, witnessUId, id_*, id_*, $\mathrm{id}_\Box$, $\mathrm{id}_\Box$, id_Δ, id_Δ]. Then unquote $\overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}}$. Result follows from Lemma C.15. □

C.2 isAbs

isAbs is defined in `lib/isAbslib.pts`. isAbs = $\lambda\alpha : \mathrm{U}.\ \lambda\mathrm{e} : \mathrm{Exp}\ \alpha$. fold[UBool, witnessUBool, $\mathrm{isAbsAbs}_*$, $\mathrm{isAbsApp}_*$, $\mathrm{isAbsAbs}_\Box$, $\mathrm{isAbsApp}_\Box$, $\mathrm{isAbsAbs}_\Delta$, $\mathrm{isAbsApp}_\Delta$] α (e UBool).

Theorem 6.3 (Type of isAbs)**.**

$$\langle\rangle \vdash \mathrm{isAbs} : (\Pi\alpha : \mathrm{U}.\ \mathrm{Exp}\ \alpha \to \mathrm{Bool})$$

Proof. From Theorem A.8, we have that isAbs : $\Pi\alpha : \mathrm{U}.\mathrm{Exp}\ \alpha \to \mathrm{UBool}\ \alpha$. The result follows from the conversion rule based on $\mathrm{UBool}\ \alpha \equiv \mathrm{Bool}$. □

Lemma C.16. witnessUBool *is an identity witness.*

Proof. Straightforward. Each witness function is an identity witness function. □

Lemma C.17. *Let f =* fold[UBool, witnessUBool, abs_*, app_*, $\mathrm{abs}_\Box$, $\mathrm{app}_\Box$, abs_Δ, app_Δ]*, and suppose* $\Gamma \vdash \mathrm{e} : \tau$.

- *If* $\mathrm{e} = \lambda x : \mathrm{A}.\mathrm{e}_1$, *then* $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta$ true.
- *If* $\mathrm{e} = \mathrm{e}_1\ \mathrm{A}$, *then* $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta$ false.

Proof. Suppose $\mathrm{e} = \lambda\mathrm{x} : \mathrm{A}.\mathrm{e}_1$. There are three cases: either $\langle\rangle \vdash \mathrm{A} : *$, or $\langle\rangle \vdash \mathrm{A} : \Box$, or $\mathrm{A} = \Box$.

Suppose $\langle\rangle \vdash \mathrm{A} : *$. Then by Theorem A.11, $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta (\lambda\mathrm{T}_1 : *.\ \lambda\mathrm{T}_2 : *.\ \lambda\mathrm{f} : \mathrm{Bool} \to \mathrm{Bool.true})\ \sigma_1\ \sigma_2\ (\lambda\mathrm{x} : \mathrm{UBool\ A}.\ \mathrm{f}\ \overline{\sigma_2}\ \overline{\mathrm{e}_1}) \equiv_\beta$ true, as required.

The cases of $\langle\rangle \vdash \mathrm{A} : \Box$ and $\mathrm{A} = \Box$ are similar to the case of $\langle\rangle \vdash \mathrm{A} : *$.

Suppose now $\mathrm{e} = \mathrm{e}_1\ \mathrm{A}$. Again there are three cases: either $\langle\rangle \vdash \mathrm{A} : \tau_1 : *$, or $\langle\rangle \vdash \mathrm{A} : \kappa : \Box$, or $\langle\rangle \vdash \mathrm{A} : \Box$.

Suppose $\langle\rangle \vdash \mathrm{A} : \tau_1 : *$. Then $\Gamma \vdash \mathrm{e}_1 : \tau_1 \to \tau_2$, and by Theorem A.11, $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{app}_*\ \tau_1\ \tau_2\ (\mathrm{f}\ \overline{\tau_1 \to \tau_2}\ \overline{\mathrm{e}_1})\ (\mathrm{f}\ \overline{\tau_1}\ \overline{\mathrm{A}}) \equiv_\beta$ false, as required.

Suppose $\langle\rangle \vdash \mathrm{A} : \kappa : \Box$. Then $\langle\rangle \vdash \mathrm{e}_1 : (\Pi\alpha : \kappa.\ \tau_1)$. By Lemma C.16 and Lemma C.12, $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{app}_\Box\ \kappa\ (\lambda\alpha : \kappa.\ \mathrm{Bool})\ (\mathrm{f}\ \overline{(\Pi\alpha : \kappa.\ \tau_1)}\ \overline{\mathrm{e}_1})\ \mathrm{A} \equiv_\beta$ false.

Suppose $\langle\rangle \vdash \mathrm{A} : \Box$. Then $\Gamma \vdash \mathrm{e}_1 : (\Pi\chi : \Box.\ \tau_1)$, and by Theorem A.11, $\mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}} \equiv_\beta \mathrm{app}_\Delta\ (\lambda\chi : \Box.\ \tau_1)\ (\mathrm{f}\ \overline{\Pi\chi : \Box.\ \tau_1}\ \overline{\mathrm{e}_1})\ \mathrm{A} \equiv_\beta$ false.

□

Theorem 6.4 (Correctness of isAbs)**.**
If $\langle\rangle \vdash \mathrm{e} : \tau : *$ *and* quote(e) = q *then:*

- *If* $\mathrm{e} = \lambda x : \mathrm{A}.\mathrm{e}_1$, *then* isAbs $\overline{\tau}\ \mathrm{q} \equiv_\beta$ true.
- *If* $\mathrm{e} = \mathrm{e}_1\ \mathrm{A}$, *then* isAbs $\overline{\tau}\ \mathrm{q} \equiv_\beta$ false.

Proof. Let f = fold[UBool, witnessUBool, abs_*, app_*, $\mathrm{abs}_\Box$, $\mathrm{app}_\Box$, abs_Δ, app_Δ]. Suppose $\langle\rangle \vdash \mathrm{e} : \tau$ and quote(e) = q. Then isAbs $\overline{\tau}\ \mathrm{q} \equiv_\beta \mathrm{f}\ \overline{\tau}\ \overline{\mathrm{e}}$. The result follows from Lemma C.17. □

C.3 cps

cps is defined in `lib/cpslib.pts`. cps = $\lambda\alpha : \mathrm{U}.\ \lambda\mathrm{e} : \mathrm{Exp}\ \alpha$. fold[CPS, witnessCPS, cpsAbs_*, cpsApp_*, $\mathrm{cpsAbs}_\Box$, $\mathrm{cpsApp}_\Box$, cpsAbs_Δ, cpsApp_Δ] α (e CPS).

Lemma C.18 (Types of CPS functions)**.**

$\langle\rangle \vdash \mathrm{CPS} : \mathrm{U} \to *$
$\langle\rangle \vdash \mathrm{witnessCPS} : \mathrm{Witness\ CPS}$
$\langle\rangle \vdash \mathrm{cpsAbs}_* : (\Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}.\ (\mathrm{CPS}\ \alpha \to \mathrm{CPS}\ \beta) \to \mathrm{CPS}\ (\mathrm{Prod}_*\ \alpha\ \beta))$
$\langle\rangle \vdash \mathrm{cpsApp}_* : (\Pi\alpha : \mathrm{U}.\ \Pi\beta : \mathrm{U}.\ \mathrm{CPS}\ (\mathrm{Prod}_*\ \alpha\ \beta) \to \mathrm{CPS}\ \alpha \to \mathrm{CPS}\ \beta)$
$\langle\rangle \vdash \mathrm{cpsAbs}_\Box : (\Pi\chi : \Box.\ \Pi\alpha : \chi \to \mathrm{U}.\ (\Pi\beta : \chi.\ \mathrm{CPS}\ (\alpha\ \beta)) \to \mathrm{CPS}\ (\mathrm{Prod}_\Box\ \chi\ \alpha))$
$\langle\rangle \vdash \mathrm{cpsApp}_\Box : (\Pi\chi : \Box.\ \Pi\alpha : \chi \to \mathrm{U}.\ \mathrm{CPS}\ (\mathrm{Prod}_\Box\ \chi\ \alpha) \to \Pi\beta : \chi.\ \mathrm{CPS}\ (\alpha\ \beta))$
$\langle\rangle \vdash \mathrm{cpsAbs}_\Delta : (\Pi\alpha : \Box \to \mathrm{U}.\ (\Pi\chi : \Box.\ \mathrm{CPS}\ \alpha\ \chi) \to \mathrm{CPS}\ (\mathrm{Prod}_\Delta\ \alpha))$
$\langle\rangle \vdash \mathrm{cpsApp}_\Delta : (\Pi\alpha : \Box \to \mathrm{U}.\ \mathrm{CPS}\ (\mathrm{Prod}_\Delta\ \alpha) \to \Pi\chi : \Box.\ \mathrm{CPS}\ (\alpha\ \chi))$

Proof. Machined checked. □

Theorem 6.5 (Type of cps)**.**

$$\langle\rangle \vdash \mathit{cps} : (\Pi\alpha : \mathit{U}.\ \mathit{Exp}\ \alpha \to \mathit{CPS}\ \alpha)$$

Proof. Follows from Lemma C.18 and Theorem A.8. □