Internalizing Relational Parametricity in the Extensional Calculus of Constructions

Neelakantan R. Krishnaswami and Derek Dreyer

Max Planck Institute for Software Systems (MPI-SWS)
Kaiserslautern and Saarbrücken, Germany <{neelk,dreyer}@mpi-sws.org>

Abstract

We give the first relationally parametric model of the extensional calculus of constructions. Our model remains as simple as traditional PER models of types, but unlike them, it additionally permits the relating of terms that implement abstract types in different ways. Using our model, we can validate the soundness of quotient types, as well as derive strong equality axioms for Church-encoded data, such as the usual induction principles for Church naturals and booleans, and the eta law for strong dependent pair types. Furthermore, we show that such equivalences, justified by relationally parametric reasoning, may soundly be internalized (*i.e.,* added as equality axioms to our type theory). Thus, we demonstrate that it is possible to interpret equality in a dependently-typed setting using parametricity. The key idea behind our approach is to interpret types as so-called *quasi-PERs* (or *zigzag-complete relations*), which enable us to model the symmetry and transitivity of equality while at the same time allowing abstract types with different representations to be equated.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Relational parametricity, dependent types, quasi-PERs

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Reynolds [21] famously introduced the concept of *relational parametricity* with a fable about data abstraction. Professors Bessel and Descartes, each teaching a class on complex numbers, defined them differently in the first lecture, the former using polar coordinates and the latter using (of course) cartesian coordinates. But despite accidentally trading sections after the first lecture, they never taught their students anything false, since after the first class, both professors proved all their theorems in terms of the defined operations on complex numbers, and never in terms of their underlying coordinate representation.

Reynolds formalized this idea by giving a semantics for System F in which each type denoted not just a set of well-formed terms, but a *logical relation* between them, defined recursively on the type structure of the language. Then, the fact that well-typed client programs were insensitive to a specific choice of implementation could be formalized in terms of their taking logically related inputs to logically related results. Since the two constructions of the complex numbers share the same interface, and it is easy to show they are logically related at that interface, any client of the interface must return equivalent results regardless of which implementation of the interface is used. Hence, parametricity gives a way of modelling a general notion of *representation-independent* program equivalence, significantly coarser (and thus more flexible) than standard notions of set-theoretic equality.

Subsequently, Plotkin and Abadi [20] showed how to build a logic in which parametricity could be used to prove the equivalence of System F programs. Plotkin-Abadi logic is a system in the LCF tradition, where an external logic is used to reason about the behavior

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of programs written in a particular programming language. This logic lets us prove, for example, that the polar and cartesian representations of the complex numbers are indeed equal at a suitable existential type.

More recently, there has been a great deal of interest in unifying programming languages with their program logic, by means of dependent type theory [19]. Dependent type systems allow types to mention program terms and thereby state strong invariants about the behavior of programs. The ability to put strong preconditions on functions means that we can support operations that might be unsafe in general (*e.g.,* unchecked array access) without compromising the safety of the programming language.

However, the notion of equality available in dependent type theory has historically been limited. In intensional type theories (*e.g.,* [2]), equality is given purely as the β-equality of terms. Even the more generous approach of extensional type theory [18, 1] still equips each set with an intrinsic notion of equality at the time of its definition. The approach of fixing a notion of equality is at odds with the "outside view" of equality suggested by relational parametricity, where equivalence is determined *relative* to the operations exported to a client. This limitation is especially galling given the recent work of Bernardy *et al.* [6], who show that the *syntax* of type theory is wholly compatible with parametricity—every well-typed term in the calculus of constructions respects the relationally parametric interpretation associated with its type—but there is no way to internalize this fact.

As a result, though it is possible to *define* many types such as existentials, coproducts, and dependent pairs, it is not possible to prove that they satisfy the expected equational properties (*e.g.,* η-rules). This is particularly frustrating in a dependently-typed setting, where being able to internalize parametricity properties would greatly facilitate verification.

In this paper, we give the first relationally parametric model of the extensional calculus of constructions [10]. Our model enables us to prove equalities and induction principles using parametricity-based reasoning, then internalize these properties as new axioms. In other words, we can *internalize relational parametricity* into our dependent type theory.

We interpret the types of the calculus of constructions by means of a logical relations model, and interpret the identity type using the relations our model defines. However, an off-the-shelf logical relation is only guaranteed to be a reflexive congruence for well-typed terms (the *fundamental property*). Logical relations in general are not necessarily symmetric or transitive, both of which are needed to use the relation as a model of *equality*.

A common approach to gaining symmetry and transitivity is to require the relations interpreting types to be partial equivalence relations (PERs), which are symmetric and transitive by definition. However, we pay a high price for mandating symmetry: relating terms with different representations (*e.g.,* Peano and binary numbers) is no longer possible, since we can no longer relate different things on each side of the relation. Consequently, we can no longer prove representation independence results in a natural way.

Traditionally, this difficulty is resolved by giving a Reynolds-style relational semantics for the language *together* with a PER model in a mutually recursive construction, and proving a correspondence between the two (the *identity extension lemma*). However, this approach more than doubles the work involved in defining a model, which hits particularly hard when facing the complex models of dependent type theory. Moreover, to our knowledge, no such parametric PER models of dependent type theory have yet been developed.

Our key innovation is to model types instead using so-called *quasi-PERs* (a.k.a. *difunctional relations*, or *zigzag-complete relations*), which generalize PERs to the asymmetric case. Using quasi-PERs, we give a single relational model which supports symmetry and transitivity of equality *as well as* the relating of terms with differing type representations.

κ	$::=$	$* \mid \Pi x : X.\ \kappa \mid \Pi\alpha : \kappa.\ \kappa'$	Kinds
X, A	$::=$	$\Pi\alpha : \kappa.\ X \mid \Pi x : X.\ Y \mid e =_X e'$	Types
	$\mid$	$\lambda x : X.\ A \mid A\ e \mid \lambda\alpha : \kappa.\ A \mid A\ B \mid \alpha$	
e	$::=$	$x \mid \lambda x : X.\ e \mid e\ e \mid \lambda\alpha : \kappa.\ e \mid e\ A \mid \mathsf{refl}$	Terms
v	$::=$	$\lambda x : X.\ e \mid \lambda\alpha : \kappa.\ e \mid \mathsf{refl}$	Values
Γ	$::=$	$\cdot \mid \Gamma, x : X \mid \Gamma, \alpha : \kappa$	Contexts

Figure 1 Syntax

$\Gamma\ \mathsf{ok}$	Context Well-formedness	$\Gamma \vdash \kappa : \mathsf{kind}$	Kind Well-formedness
$\Gamma \vdash A : \kappa$	Well-kinding of Type Constructors	$\Gamma \vdash e : X$	Well-typing of Expressions
$\Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}$	Definitional Equality for Kinds	$\Gamma \vdash A \equiv A' : \kappa$	Definitional Equality for Types
$\Gamma \vdash e \equiv e' : X$	Definitional Equality for Terms	$e \mapsto e'$	Operational Semantics

Figure 2 Summary of Judgments

To illustrate this point, we show how to state parametricity axioms for some types familiar from System F such as Church numerals and existentials. Exploiting the presence of type dependency, we also show how parametricity can be used to recover strong dependent pairs (*i.e.*, $\Sigma x : X.\ Y$ with π_1 and π_2 projections) from a Church encoding, as well as showing the soundness of quotient types in our model. Proof details can be found in the accompanying technical appendix, online at `<http://www.mpi-sws.org/~dreyer/irpecc-appendix.pdf>`.

2 Syntax, Typing and Operational Semantics

Our overall system is an explicitly-typed version of the calculus of constructions, extended with an identity type and an elimination rule for equality based on equality reflection. We use extensional type theory to make equality axioms (e.g., η for Church encodings of pairs) behave well. In an intensional system, equality axioms make subject reduction fail, since the eliminator for the equality type gets stuck. Extensional type theory makes equality elimination implicit, and so has better computational behavior. In Figure 1, we give the syntactic categories of our type system, and in Figure 2, we list the judgements in our system. Most of the kinding and typing rules are standard, as is the standard call-by-name evaluation semantics, and so we omit them for space (they can be found in the appendix).

We present our system with distinct syntactic categories for kinds (ranged over by metavariables κ), types (ranged over by metavariables X, Y, A, B, C, and type variables α, β) and terms (ranged over by metavariables e, and term variables x, y, z). We typically adopt the convention of using A, B, and C for type constructors of arbitrary kind, and X and Y for type constructors of base kind. Almost all of the typing rules are standard for the calculus of constructions, and we discuss only our variations in detail.

Our treatment of equality follows extensional type theory. We introduce identities with refl when two terms are equal, and so the definitional equality for identity types satisfies the uniqueness of identity proofs property (a.k.a. Axiom K [15]). We also support equality reflection: if $\Gamma \vdash e_p : e =_X e'$, then $\Gamma \vdash e \equiv e' : X$. This rule makes typechecking undecidable, as typing derivations may need to invent equality proofs. To summarize, our definitional equality is just the $\beta\eta$-theory of the lambda calculus plus rules for equality types. For expository reasons, we have not included any parametricity properties in definitional equality, saving these for Section 5.

3 Semantics

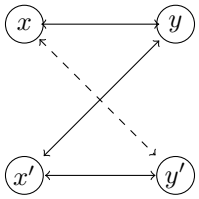
Our overall semantics is a realizability model, in which types are interpreted as relations between closed expressions e. However, since the syntactic types appearing within expressions are computationally irrelevant, we simplify matters by working with relations over **Exp**, the set of equivalence classes of closed expressions modulo differences in syntactic types. That is, in the model, we consider $\lambda x : X.\ e = \lambda x : Y.\ e$, $\lambda \alpha : \kappa.\ e = \lambda \alpha : \kappa'.\ e$, and $e\ A = e\ B$, for *arbitrary* X, Y, κ, κ', A, B. This is analogous to building the model with type-erased terms, and we will sometimes write _ in place of (irrelevant) type annotations and arguments.

3.1 Quasi-PERs

The primary technical innovation in our work is to switch from a PER semantics of types to an interpretation based on *quasi-PERs*, which generalize PERs to the asymmetric case.

▶ **Definition 1.** (Quasi-PER) A quasi-PER between two sets $\hat{X}$ and $\hat{Y}$ is a *zigzag-complete* relation $R \subseteq \hat{X} \times \hat{Y}$: if $(x, y) \in R$, and $(x', y') \in R$ and $(x', y) \in R$, then $(x, y') \in R$.

The zigzag condition is best visualized pictorially:



So if R tells us that two elements of $\hat{X}$ are related to a given y, then they are related to all the same elements of $\hat{Y}$. Indeed, given a QPER $R \subseteq \hat{X} \times \hat{Y}$, both $R \circ R^{-1}$ is a PER on $\hat{X}$, and $R^{-1} \circ R$ is a PER on $\hat{Y}$. (All QPERs arise in this way, and so this could equivalently be taken as a definition of quasi-PERs.)

Like PERs, QPERs form a complete lattice. The meet of two QPERs is the intersection, and indeed they are are closed under arbitrary intersections. As a result, they also have arbitrary joins, with the join $\bigsqcup \mathcal{R}$ defined as the intersection of every QPER containing $\bigcup \mathcal{R}$. The join will, in general, have more elements that the union, best illustrated by the direct construction of the join:

$$\begin{array}{lcl} (\bigsqcup \mathcal{R})_0 & = & \bigcup \mathcal{R} \\ (\bigsqcup \mathcal{R})_{k+1} & = & (\bigsqcup \mathcal{R})_k \cup \{(x, y') \mid (x, y) \in (\bigsqcup \mathcal{R})_k \ \wedge\ (x', y') \in (\bigsqcup \mathcal{R})_k \ \wedge\ (x', y) \in (\bigsqcup \mathcal{R})_k\} \\ \bigsqcup \mathcal{R} & = & \bigcup_{k \in \mathbb{N}} (\bigsqcup \mathcal{R})_k \end{array}$$

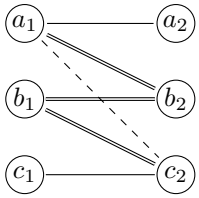
So the join takes the union and fills in all the missing zigzags.

Our reason for using QPERs in our semantics of types is quite simple. When giving a relational type interpretation, we are pulled in two contrary directions. First, we want to use the relation to model representation independence: we want to be able to say that two different implementations of the same interface are equal. For this, we need to consider relations between different sets. Second, we also want to use our logical relation to *define* equality at each type. For this, we (seemingly) need symmetry and transitivity properties on the relations we use to interpret types. These apparently conflicting demands can be reconciled by interpreting types as QPERs. While QPERs are capable of relating terms of different types, they also induce a canonical equivalence relation, which we can use in turn to model equality. The trick is that this equivalence relation is not a relation on terms, but on *pairs of related terms*.

To see how this works, first suppose we have two well-typed terms e and t at some type X. The fundamental property of logical relations will tell us that $(e_1, e_2) \in [\![X]\!]$ and $(t_1, t_2) \in [\![X]\!]$, where (e_1, e_2) and (t_1, t_2) are essentially e and t under different, but related, environments. The model of $e \equiv t : X$ will then be the proposition that $(e_1, t_2) \in [\![X]\!]$.

For the relation to model symmetry, *i.e.,* that $e \equiv t$ implies $t \equiv e$ for $e, t : X$, we will need to show that $(e_1, t_2) \in [\![X]\!]$ (together with the knowledge that (e_1, e_2) and (t_1, t_2) are in $[\![X]\!]$, thanks to the fundamental property) implies $(t_1, e_2) \in [\![X]\!]$. But this is *precisely* the definition of zigzag closure!

Similarly, we can show that transitivity holds for well-typed terms. Consider the diagram:



Here, $(a_1, a_2), (b_1, b_2)$, and (c_1, c_2) can again be thought of as programs a, b, c under different (but related) environments. Given that (a_1, b_2) and (b_1, c_2) are in $[\![X]\!]$, the zigzag closure lets us derive $(a_1, c_2) \in [\![X]\!]$, as required for transitivity.

Consequently, each QPER $Q \subseteq R \times S$ may also be viewed as a PER on the set $R \times S$:

▶ **Definition 2.** (Canonically induced PER) Every QPER $Q \subseteq R \times S$ induces an equivalence relation $\sim_Q \subseteq Q \times Q$ (and hence a PER on $R \times S$), defined as $(a_1, a_2) \sim_Q (b_1, b_2)$ iff the zigzag $\{(a_1, a_2), (b_1, b_2), (a_1, b_2), (b_1, a_2)\} \subseteq Q$. (We write $\sim$ as shorthand for $\sim_Q$ if Q is evident from context.)

Hence, a QPER provides a way of telling when pairs of related terms are equivalent. In this sense, our approach reverses the usual method of building parametric models. Instead of building a PER model of types *as well as* a relational model between such PERs, we use QPERs to directly define a relational model, from which a canonical PER model can then be derived after the fact.

3.2 A Comparison with PER Models

Readers familiar with traditional parametric models may be surprised with the QPER structure: since the composition of a QPER with itself is a PER, shouldn't QPERs have merely the same expressive power as PERs? In fact, QPERs give rise to a significantly coarser notion of program equivalence than PERs.

Traditional PER models are symmetric, and therefore cannot relate different implementations of the same type (e.g., PER models cannot show that Peano and binary representations of the natural numbers are equivalent). This deficiency is then rectified by giving a second relational model. However, the asymmetry inherent in QPERs allows us to relate different implementations, *without* having to give two models — indeed, our model seems to validate all of the program equivalences provable in Abadi-Plotkin logic.

3.3 Contexts

The interpretation of the Γ ok judgment is the *set of grounding environments* γ that satisfy it. We give the interpretation in Figure 4.

An environment γ is in the interpretation of the empty context iff it is the empty environment. It is in the interpretation of the context $\Gamma, x : X$ ok iff it is an element of $[\![\Gamma \text{ ok}]\!]$,

$$\textsc{Cand} \triangleq \left\{ R \in \textsc{Qper}(\mathrm{Exp}, \mathrm{Exp}) \;\middle|\; \begin{array}{l} \forall (e_1, e_2) \in R.\ e_1 \downarrow \wedge\ e_2 \downarrow\ \wedge \\ \forall (e_1, e_2) \in R, (e_1', e_2') \in \mathrm{Exp}^2. \\ \qquad e_1 \leftrightarrow^* e_1' \wedge e_2 \leftrightarrow^* e_2' \implies (e_1', e_2') \in R \end{array} \right\}$$

Figure 3 Candidate Relations

together with a pair (e, e') of closed terms from the interpretation of $\Gamma \vdash X : *$. Finally, γ is in the interpretation of the context $\Gamma, \alpha : \kappa$ ok iff it is an element of $[\![\Gamma\ \mathsf{ok}]\!]$, together with a tuple $((A, A'), R)$. Here, A and A' are closed syntactic types, and R is the semantic interpretation of the type. Note that there are no well-formedness constraints on the *syntactic* types A and A': we do not need them, since the operational semantics never examines a type constructor, and the relation R carries all the necessary semantic constraints.

In Figure 4, we also define a notion of equivalence $\gamma \sim_{\Gamma\ \mathsf{ok}} \gamma'$ on environments. This relation says that the relations R and R' to which γ and γ' map the same type variable α must be equal, and that pairs of terms $(e_1, e_1')/x$ and $(e_2, e_2')/x$ must lie in the same equivalence class of the relation. The $\sim_{\Gamma\ \mathsf{ok}}$ relation is indeed a PER, but actually proving that fact can only be done after the proof of soundness of the interpretation of types and kinds. This is due to the fact that the definition is "biased"—the second line asymmetrically uses $[\![\Gamma \vdash X : *]\!]\ \gamma_1$ on the right-hand side.

In Figure 5, we give notation concerning environments that we will use in the sequel. γ contains left- and right-bindings for each of the variables in its domain; γ_1 is the left projection of the environment, and γ_2 is the right projection. We write $\gamma(e)$ to indicate the pair of terms we get from the left and right projections of γ applied to e.

3.4 Kinds

We give the semantics of kinds in Figure 7. We begin by giving a "pre-interpretation" function $\|\cdot\|$ (defined in Figure 6), which gives an approximate interpretation of kinds, without reference to term or type arguments. This interpretation is less precise than we want, but is a useful device to simplify the argument that our main interpretation function $[\![\cdot]\!]$ is well-defined. That main interpretation of kinds, $[\![\Gamma \vdash \kappa : \mathsf{kind}]\!]$, on the other hand, *is* relative to a context γ. The interpretation of the base kind $\Gamma \vdash * : \mathsf{kind}$, given in Figure 3, is a slight restriction of the set of quasi-PERs on expressions. Namely, we restrict ourselves to quasi-PERs of *terminating* expressions, closed under expansion and reduction.

The interpretation of the higher kind $\Gamma \vdash \Pi\alpha : \kappa.\ \kappa' : \mathsf{kind}$ is morally a currying of the interpretation $\Gamma, \alpha : \kappa \vdash \kappa' : \mathsf{kind}$. In particular, it is the set of functions $\|\kappa\| \to \|\kappa'\|$, such that (1) we ignore the syntactic part of an argument triple $((X_1, X_2), R)$, and (2) on any argument $R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma$, the result is in $[\![\Gamma, \alpha : \kappa \vdash \kappa' : \mathsf{kind}]\!]\ (\gamma, ((A_1, A_2), R)/\alpha)$. On anything outside the dependent domain, we force the result to be a fixed ("dummy") element $!_{\kappa'} \in \|\kappa'\|$. Similarly, elements of $\Gamma \vdash \Pi x : X.\ \kappa' : \mathsf{kind}$ are the currying of the interpretation $\Gamma, x : X \vdash \kappa' : \mathsf{kind}$, with the condition that elements return the same result for all equivalent pairs $(e_1, e_2) \sim_{\hat{X}} (e_1', e_2')$, where $\hat{X} = [\![\Gamma \vdash X : *]\!]\ \gamma$ is the relational interpretation of X.

3.5 Type Constructors

In Figure 8, we give the interpretation of the type constructors of our language, as a function that takes a kinding derivation and returns an element of the appropriate semantic kind. The first line of the definition says that the interpretation of $\Gamma \vdash \alpha : \kappa$ proceeds by looking up α in the environment argument γ, and returning the relation component of the triple. (Here and elsewhere, we use overbar notation to denote pairs, *e.g.,* $\overline{A}$ denotes (A, A').)

$$
\begin{array}{lcl}
[\![\Gamma\ \mathsf{ok}]\!] & \in & \mathcal{P}(\|\Gamma\|) \\
[\![\cdot\ \mathsf{ok}]\!] & = & \{\langle\rangle\} \\
[\![\Gamma, x : X\ \mathsf{ok}]\!] & = & \{(\gamma, (e, e')/x) \mid \gamma \in [\![\Gamma\ \mathsf{ok}]\!] \wedge (e, e') \in [\![\Gamma \vdash X : *]\!]\ \gamma\} \\
[\![\Gamma, \alpha : \kappa\ \mathsf{ok}]\!] & = & \{(\gamma, ((X, X'), R)/\alpha) \mid \gamma \in [\![\Gamma\ \mathsf{ok}]\!] \wedge ((X, X'), R) \in (\mathrm{Type} \times \mathrm{Type}) \times [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma\}
\end{array}
$$

$$
\begin{array}{lcl}
\langle\rangle \sim_{\cdot\ \mathsf{ok}} \langle\rangle & \iff & \text{always} \\
(\gamma_1, (e_1, e_1')/x) \sim_{(\Gamma, x:X\ \mathsf{ok})} (\gamma_2, (e_2, e_2')/x) & \iff & \gamma_1 \sim_{\Gamma\ \mathsf{ok}} \gamma_2 \wedge (e_1, e_1') \sim_{[\![\Gamma \vdash X:*]\!]\ \gamma_1} (e_2, e_2') \\
(\gamma_1, (\overline{A}, R_1)/\alpha) \sim_{(\Gamma, \alpha:\kappa\ \mathsf{ok})} (\gamma_2, (\overline{B}, R_2)/\alpha) & \iff & \gamma_1 \sim_{\Gamma\ \mathsf{ok}} \gamma_2 \wedge R_1 = R_2
\end{array}
$$

Figure 4 Interpretations of Contexts and Environment Equivalence

$$
\begin{array}{lcllcl}
(\langle\rangle)_i & = & \langle\rangle & \gamma(e) & = & (\gamma_1(e), \gamma_2(e)) \\
(\gamma, (e_1, e_2)/x)_i & = & \gamma_i, e_i/x & \gamma(A) & = & (\gamma_1(A), \gamma_2(A)) \\
(\gamma, ((A_1, A_2), R)/\alpha)_i & = & \gamma_i, A_i/\alpha & \gamma(\kappa) & = & (\gamma_1(\kappa), \gamma_2(\kappa))
\end{array}
$$

Figure 5 Notation

$$
\begin{array}{lcl}
\|*\| & = & \mathrm{Rel}(\mathrm{Exp}, \mathrm{Exp}) \\
\|\Pi x : X.\ \kappa\| & = & \mathrm{Exp}^2 \to \|\kappa\| \\
\|\Pi\alpha : \kappa.\ \kappa'\| & = & (\mathrm{Type}^2 \times \|\kappa\|) \to \|\kappa'\|
\end{array}
\qquad
\begin{array}{lcl}
\|\cdot\| & = & \{\langle\rangle\} \\
\|\Gamma, x : X\| & = & \left\{(\gamma, (e, e')/x) \;\middle|\; \gamma \in \|\Gamma\| \wedge (e, e') \in \mathrm{Exp}^2\right\} \\
\|\Gamma, \alpha : \kappa\| & = & \left\{(\gamma, ((A, A'), R)/\alpha) \;\middle|\; \begin{array}{l} \gamma \in \|\Gamma\| \wedge \\ (A, A') \in \mathrm{Type}^2 \wedge \\ R \in \|\kappa\| \end{array}\right\}
\end{array}
$$

$$
\begin{array}{lcl}
!_* & = & \emptyset \\
!_{\Pi x:X.\ \kappa} & = & \lambda(e, e') \in \mathrm{Exp}^2.\ !_\kappa \\
!_{\Pi\alpha:\kappa.\ \kappa'} & = & \lambda(\overline{A}, R) \in \mathrm{Type}^2 \times \|\kappa\|.\ !_{\kappa'}
\end{array}
$$

Figure 6 Pre-Interpretations of Kinds and Contexts

The interpretation of a lambda-abstraction $\lambda\alpha : \kappa.\ A$ is just a function that takes an argument in κ, and returns the result of interpreting A in an extended environment. Likewise, a type constructor application $A\ B$ takes the meaning of A, and passes it the syntax and semantics of B. Similarly, a term abstraction $\lambda x : X.\ A$ just returns a function which takes a pair (e, e'), and returns the interpretation of A in an extended environment, and application $A\ e$ passes $\gamma(e)$, the pair of related instantiations of e, to the interpretation of A.

When the kind conversion rule is used to replace the kind of the constructor with an equivalent one, we simply interpret the subderivation and return that as our answer. As a result, however, we need an easy-to-prove coherence property for our semantic interpretations, stating that the interpretation of A is the same at any kind it inhabits (see the appendix).

Next, we give the interpretations of types of base kind. The kinding interpretation requires that such types be interpreted as relations (specifically, QPERs) between expressions. The interpretation of the function type $\Pi x : X.\ Y$ is the set of (terminating) expressions that take related arguments in X to related results in Y, in the context extended by the argument pair. This is essentially the usual rule for function types in logical relations, adjusted to support dependency. The interpretation of the polymorphic type $\Pi\alpha : \kappa.\ X$ puts a pair of type abstractions in the relation, if for each relation R in kind κ, their bodies are related at the expression relation for X, in the environment augmented with R for α.

A pair of terms reducing to $(\mathsf{refl}, \mathsf{refl})$ inhabit the identity type $e_1 =_X e_2$ only when $\gamma_1(e_1)$ and $\gamma_2(e_2)$ are related at X (alternatively, $\gamma(e_1) \sim_{\hat{X}} \gamma(e_2)$). Since the identity type is interpreted by a relation containing at most one pair of values, we validate axiom K.

$$[\![\Gamma \vdash \kappa : \mathsf{kind}]\!] \in \|\Gamma\| \to \mathcal{P}(\|\kappa\|)$$

$$\begin{array}{lcl}
[\![\Gamma \vdash * : \mathsf{kind}]\!]\ \gamma & = & \textsc{Cand} \\
[\![\Gamma \vdash \Pi\alpha : \kappa.\ \kappa' : \mathsf{kind}]\!]\ \gamma & = & \left\{ T \in \|\Pi\alpha : \kappa.\ \kappa'\| \;\middle|\; \begin{array}{l} \forall \overline{A}, \overline{B}, R \in \|\kappa\|\,.\ T(\overline{A}, R) = T(\overline{B}, R) \wedge \\ \forall \overline{A}, R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma. \\ \quad T(\overline{A}, R) \in [\![\Gamma, \alpha : \kappa \vdash \kappa' : \mathsf{kind}]\!]\ (\gamma, (\overline{A}, R)/\alpha) \wedge \\ \forall \overline{A}, R \notin [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma.\ T(\overline{A}, R) = !_{\kappa'} \end{array} \right\} \\
[\![\Gamma \vdash \Pi x : X.\ \kappa : \mathsf{kind}]\!]\ \gamma & = & \mathsf{let}\ \hat{X} = [\![\Gamma \vdash X : *]\!]\ \gamma\ \mathsf{in} \\
 & & \left\{ R \in \|\Pi x : X.\ \kappa\| \;\middle|\; \begin{array}{l} \forall \bar{e}, \overline{e'} \in \hat{X}.\ \bar{e} \sim_{\hat{X}} \overline{e'} \implies R\ \bar{e} = R\ \overline{e'} \wedge \\ \forall \bar{e} \in \hat{X}.\ R\ \bar{e} \in [\![\Gamma, x : X \vdash \kappa : \mathsf{kind}]\!]\ (\gamma, \bar{e}/x) \wedge \\ \forall \bar{e} \notin \hat{X}.R\ \bar{e} = !_{\kappa} \end{array} \right\}
\end{array}$$

Figure 7 Interpretations of Kinds

$$[\![\Gamma \vdash A : \kappa]\!] \in \|\Gamma\| \to \|\kappa\|$$

$$\begin{array}{lcl}
[\![\Gamma, \alpha : \kappa, \Gamma' \vdash \alpha : \kappa]\!]\ \gamma & = & R \qquad \text{if } \gamma(\alpha) = (\overline{A}, R) \\
[\![\Gamma \vdash \lambda\alpha : \kappa.\ A : \Pi\alpha : \kappa.\ \kappa']\!]\ \gamma & = & \lambda(\overline{B}, R).\ \begin{cases} [\![\Gamma, \alpha : \kappa \vdash A : \kappa']\!]\ (\gamma, (\overline{B}, R)/\alpha) & \text{if } R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma \\ !_{\kappa'} & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash A\ B : [B/\alpha]\kappa']\!]\ \gamma & = & [\![\Gamma \vdash A : \Pi\alpha : \kappa.\ \kappa']\!]\ \gamma\ (\gamma(B), [\![\Gamma \vdash B : \kappa]\!]\ \gamma) \\
[\![\Gamma \vdash \lambda x : X.\ A : \Pi x : X.\ \kappa]\!]\ \gamma & = & \lambda\bar{e}.\ \begin{cases} [\![\Gamma, x : X \vdash A : \kappa]\!]\ (\gamma, \bar{e}/x) & \text{if } \bar{e} \in [\![\Gamma \vdash X : *]\!]\ \gamma \\ !_{\kappa} & \text{otherwise} \end{cases} \\
[\![\Gamma \vdash A\ e : [e/x]\kappa]\!]\ \gamma & = & [\![\Gamma \vdash A : \Pi x : X.\ \kappa]\!]\ \gamma\ \gamma(e) \\
[\![\Gamma \vdash A : \kappa]\!]\ \gamma & = & [\![\Gamma \vdash A : \kappa']\!]\ \gamma\ (\text{when } \Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind})
\end{array}$$

$$[\![\Gamma \vdash \Pi x : X.\ Y : *]\!]\ \gamma = \left\{ (e_1, e_1') \;\middle|\; \begin{array}{l} e_1 \downarrow \wedge\ e_1' \downarrow \wedge \\ \forall\, (e_2, e_2') \in [\![\Gamma \vdash X : *]\!]\ \gamma. \\ \quad (e_1\ e_2, e_1'\ e_2') \in [\![\Gamma, x : X \vdash Y : *]\!]\ (\gamma, (e_2, e_2')/x) \end{array} \right\}$$

$$[\![\Gamma \vdash \Pi\alpha : \kappa.\ X : *]\!]\ \gamma = \left\{ (e, e') \;\middle|\; \begin{array}{l} e \downarrow \wedge\ e' \downarrow \wedge \\ \forall A, A', R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma. \\ \quad (e\ A, e'\ A') \in [\![\Gamma, \alpha : \kappa \vdash X : *]\!]\ (\gamma, ((A, A'), R)/\alpha) \end{array} \right\}$$

$$[\![\Gamma \vdash e_1 =_X e_2 : *]\!]\ \gamma = \{(e, e') \mid e \mapsto^* \mathsf{refl} \wedge e' \mapsto^* \mathsf{refl} \wedge (\gamma_1(e_1), \gamma_2(e_2)) \in [\![\Gamma \vdash X : *]\!]\ \gamma\}$$

Figure 8 Interpretations of Type Constructors

4 Soundness

Our main theorem is a proof of Reynolds' fundamental property for our language. By induction over derivations, we can show that every well-typed expression is related to itself by the relational interpretation of its type. Our proof proceeds in two stages.

1. Using the pre-interpretation of contexts and kinds given in Figure 6, which are clearly well-defined, we first show basic structural properties of the main semantic interpretation—namely, that it is well-defined, that it is coherent, and that it satisfies semantic weakening and substitution properties. (For space reasons, we do not state the exact lemmas in this extended abstract, but the lemmas and their proofs can be found in the appendix.)

2. Then, we prove the fundamental property. This is a large structural induction over the syntax of kind, type and term derivations, as well as equality derivations. We state the fundamental property below, and give the complete proof in the appendix.

4.1 The Pre-Interpretation and Structural Properties

The pre-interpretation of kinds $\|\kappa\|$ (Figure 6) interprets κ as a set, by induction on the syntax of κ, ignoring term and type indices. The pre-interpretation $\|\Gamma\|$ merely characterizes the *shape* of environments that semantically realize Γ, without placing any interesting invariants on them. This turns out to be sufficient for "bootstrapping" purposes (*i.e.,* for defining the main semantic interpretations $[\![\cdot]\!]$), as well as for proving various properties like semantic weakening and substitution. By virtue of their being hygienic and structural, these properties hold under the assumption that the environments they quantify over merely belong to the *pre*-interpretation of contexts rather than the main interpretation. As a result, we can establish these properties independently of the fundamental theorem, and thus rely on them freely (not just inductively) when proving the fundamental theorem.

4.2 Fundamental Property

Like the structural properties, we show the fundamental theorem by an inductive case analysis of the typing derivation. Unlike the structural properties, the fundamental theorem *does* require that environments γ be semantically well-formed (*i.e.,* are elements of $[\![\Gamma\ \mathsf{ok}]\!]$). Furthermore, we will need to show that the interpretations are appropriately invariant with respect to environment equivalence ($\gamma \sim_{\Gamma\ \mathsf{ok}} \gamma'$). Since there are many different judgment forms, we have to give clauses for each judgment form.

▶ **Theorem 3** (Fundamental Property)**.**
Suppose $\Gamma\ \mathsf{ok}$*, and* $\gamma, \gamma' \in [\![\Gamma\ \mathsf{ok}]\!]$ *such that* $\gamma \sim \gamma'$*. Then:*

1. *If* $D :: \Gamma \vdash \kappa : \mathsf{kind}$*, then* $[\![D]\!]\ \gamma = [\![D]\!]\ \gamma'$*.*
2. *If* $D :: \Gamma \vdash A : \kappa$*, then* $[\![D]\!]\ \gamma = [\![D]\!]\ \gamma'$*.*
3. *If* $D :: \Gamma \vdash e : X$ *then* $\gamma(e) \sim \gamma'(e) \in [\![\Gamma \vdash X : *]\!]\ \gamma$*.*
4. *If* $D :: \Gamma \vdash A : \kappa$*, then* $[\![D]\!]\ \gamma \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma$*.*
5. *If* $\Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}$*, then* $[\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma = [\![\Gamma \vdash \kappa' : \mathsf{kind}]\!]\ \gamma'$*.*
6. *If* $\Gamma \vdash A \equiv A' : \kappa$*, then* $[\![\Gamma \vdash A : \kappa]\!]\ \gamma = [\![\Gamma \vdash A' : \kappa]\!]\ \gamma'$*.*
7. *If* $\Gamma \vdash e_1 \equiv e_2 : X$*, then* $\gamma(e_1) \sim \gamma'(e_2) \in [\![\Gamma \vdash X : *]\!]\ \gamma$*.*

This theorem justifies the use of parametricity reasoning about our language, since all well-typed terms are self-related by the corresponding relational interpretations of types. As a corollary, parametricity implies consistency: since the relational interpretation of the type $\Pi\alpha : *.\ \alpha$ is empty, it must also be a syntactically uninhabited type.

5 Examples

In all of the following proofs we assume Γ is well-formed, and that environment γ inhabits $[\![\Gamma\ \mathsf{ok}]\!]$ in order to appeal to the fundamental property.

5.1 Sums and Natural Numbers

Recall the Church encodings of some the basic data types in System F — the empty type 0 as $\Pi\alpha : *.\ \alpha$, the sum type $A + B$ as $\Pi\alpha : *.\ (A \to \alpha) \to (B \to \alpha) \to \alpha$, and the natural numbers $\mathbb{N}$ as $\Pi\alpha : *.\ \alpha \to (\alpha \to \alpha) \to \alpha$. Our model validates the expected β and η properties for these types. The proofs follow on the standard lines, and so we leave the details to the appendix. It is also possible to use parametricity to internalize the induction principles. This is more convenient to state with dependent records, so we will give that example next.

5.2 Dependent Records

Cartesian products can be defined in the usual way, and there are no surprises with them. More interesting is the fact that dependent records (Σ-types) are realizable in our model.

$$\Sigma x : X.\ Y \triangleq \Pi\alpha : *.\ (\Pi x : X.\ Y \to \alpha) \to \alpha$$

with the introduction form:

$$\mathsf{pair}\ x\ y \triangleq \lambda\alpha : *.\ \lambda k : \Pi x : X.\ Y \to \alpha.\ k\ x\ y$$

However, when it comes to eliminators, this type looks like a *weak* pair type, corresponding to a type with an eliminator $\mathsf{let}\ (x, y) = p\ \mathsf{in}\ e'$, rather than projective eliminators like $\pi_1(p)$ and $\pi_2(p)$. In the absence of parametricity, this is correct, but it is a remarkable fact [11] that in a parametric model, we can realize *strong* eliminators for this type, defined as follows:

- $\mathsf{fst} : (\Sigma x : X.\ Y) \to X = \lambda p.\ p\ X\ (\lambda x.\ \lambda y.\ x)$
- $\mathsf{snd} : \Pi p : (\Sigma x : X.\ Y).\ [\mathsf{fst}\, p/x]Y = \lambda p.\ p\ (\Sigma x : X.\ Y)\ \mathsf{pair}\ ([\mathsf{fst}\, p/x]Y)\ (\lambda x.\ \lambda y.\ y)$

Note that the projective eliminator snd is *not* syntactically well-typed. Instead, we will use our parametric model to show that it has the correct *semantic* type and equations, and so it *realizes* the projective eliminator. This means it is safe to add as an axiom to our system, and that it will have good computational behavior.

▶ **Lemma 4.** *(Normal forms for eta-expanded pairs)*
If $(p, p') \in [\![\Gamma \vdash \Sigma x : X.\ Y : *]\!]\ \gamma$*, then there exist terms* u, u', t, t' *such that* $(u, u') \in [\![\Gamma \vdash X : *]\!]\ \gamma$ *and* $(t, t') \in [\![\Gamma, x : X \vdash Y : *]\!]\ (\gamma, (u, u')/x)$ *such that* $p\ \_\ \mathsf{pair} \leftrightarrow^* \mathsf{pair}\ u\ t$ *and* $p'\ \_\ \mathsf{pair} \leftrightarrow^* \mathsf{pair}\ u'\ t'$.

▶ **Lemma 5.** *(Weak eta for pairs)*
If $(p, p') \in [\![\Gamma \vdash \Sigma x : X.\ Y : *]\!]\ \gamma$*, then* $(p, p') \sim (p\ \_\ \mathsf{pair}, p'\ \_\ \mathsf{pair})$.

The proofs of these two lemmas are essentially standard, and together imply that this type correctly encodes a weak pair. We use these facts to show snd has the correct semantic type:

▶ **Lemma 6.** *(Semantic well-typedness for* snd*)*
We have that $(\mathsf{snd}, \mathsf{snd}) \in [\![\Gamma \vdash \Pi q : (\Sigma x : X.\ Y).\ [\mathsf{fst}\, q/x]Y : *]\!]\ \gamma$.

This proof is direct, but it relies critically on the context and environment being well-formed, since it appeals to the fundamental property in several places.

▶ Proposition 7. (Projective eta for Σ-types)
If $(p, p') \in [\![\Gamma \vdash \Sigma x : X.\ Y : *]\!]\ \gamma$, then $(p, p') \sim (\mathsf{pair}\ (\mathsf{fst}\, p)\ (\mathsf{snd}\, p), \mathsf{pair}\ (\mathsf{fst}\, p')\ (\mathsf{snd}\, p'))$.
This follows easily with the previous three lemmas in hand.

5.3 Induction for the Natural Numbers

Though Church numerals are directly definable with polymorphism, their induction principle is not. That is, there is no syntactically typable term

$$\mathit{ind} : \Pi P : \mathbb{N} \to *.\ P(z) \to (\Pi n : \mathbb{N}.\ P(n) \to P(s\ n)) \to \Pi n : \mathbb{N}.\ P(n)$$

However, we can show that this term is realizable within our model. That is, we can show that the term

$$\begin{array}{ll} \lambda P, i, f, n. & \mathsf{let}\ o = \mathsf{pair}\ z\ i\ \mathsf{in} \\ & \mathsf{let}\ h = \lambda p.\ \mathsf{pair}\ (s\ (\mathsf{fst}\, p))\ (f\ (\mathsf{fst}\, p)\ (\mathsf{snd}\, p))\ \mathsf{in} \\ & \mathsf{snd}\ (n\ (\Sigma x : \mathbb{N}.\ P(x))\ o\ h) \end{array}$$

is related to itself at the type above. By using a dependent pair, we we can package up the two arguments of $\Pi n : \mathbb{N}.\ P(n) \to P(s\ n)$ into a single argument, which is what the step function in the Church encoding expects. We then use parametricity to prove that for all n, applying the iterator $n\ (\Sigma x : \mathbb{N}.\ P(x))\ o\ h$ gives us a record whose first component is n, and so whose second component must be of type $P(n)$. Details are given in the appendix.

5.4 Existential Types

Our model supports the standard encoding of existential types:

$$\begin{array}{lcl} \exists \alpha : \kappa.\ X(\alpha) & \triangleq & \Pi \beta : *.\ (\Pi \alpha : \kappa.\ X(\alpha) \to \beta) \to \beta \\ \mathsf{pack} & : & \Pi \alpha : \kappa.\ X(\alpha) \to \exists \alpha : \kappa.\ X(\alpha) \\ \mathsf{pack} & = & \lambda \alpha : \kappa.\ \lambda x.\ (\lambda \beta : *.\ \lambda k.\ k\ \alpha\ x) \end{array}$$

We can easily validate the expected β and η laws for existentials, as well as the representation independence principle, which allows existential packages with different but related implementations to be proven equivalent. Again, the proofs are standard (e.g., see [3]), and we leave them for the appendix. More interestingly, and perhaps surprisingly, we can show the soundness of an existential equality principle similar to the one from Plotkin-Abadi logic (left-to-right direction of Theorem 7 of [20]):

▶ Proposition 8. (Existential equality) If $(e, e') \in [\![\Gamma \vdash \exists \alpha : \kappa.\ X : *]\!]\ \gamma$, then there exist

1. $A, A' \in \text{Type}$,
2. $R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma$
3. $(t, t') \in [\![\Gamma, \alpha : \kappa \vdash X : *]\!]\ (\gamma, ((A, A'), R)/\alpha)$

such that $(e, e') \sim_{[\![\Gamma \vdash \exists \alpha : \kappa.\ X : *]\!]\ \gamma} (\mathsf{pack}\ A\ t, \mathsf{pack}\ A'\ t')$.

This says that any two terms (e, e') related at existential type must be equivalent to *some* packages $(\mathsf{pack}\ A\ t, \mathsf{pack}\ A'\ t')$ that are related by a representation independence argument.

Proof. First consider the pair (e, e'), and the application $(e\ \_,\ e'\ \_)$. Starting from $(e, e') \in [\![\Gamma \vdash \exists \alpha : \kappa.\ X : *]\!]\ \gamma$, we instantiate the type abstraction on both sides and choose the relational interpretation of the abstract type to be the following, defined by a QPER join:

$$S \triangleq \bigsqcup_{R \in [\![\kappa]\!]\ \gamma} \left\{ (\mathsf{pack}\ A\ e, \mathsf{pack}\ A'\ e') \;\middle|\; \begin{array}{l} (A, A') \in \text{Type}^2 \wedge \\ (e, e') \in [\![\Gamma, \alpha : \kappa \vdash X : *]\!]\ (\gamma, ((A, A'), R)/\alpha) \end{array} \right\}^{\dagger}$$

We write $T^{\dagger}$ to indicate the reduction and expansion-closure of a relation on terms T. Next, we verify that $(\mathsf{pack}, \mathsf{pack}) \in [\![\Gamma, \beta : * \vdash \Pi \alpha : \kappa.\ X \to \beta : *]\!]\ (\gamma, (\_, S)/\beta)$. From this we get:

$$(e\ \_\ \mathsf{pack},\ e'\ \_\ \mathsf{pack}) \in S$$

Note that these terms are eta-expansions of (e, e'), which must therefore be equivalent to $(\mathsf{pack}\ \_\ s, \mathsf{pack}\ \_\ s')$ for some s and s'.

Ideally, we would like to use the fact that $(\mathsf{pack}\ \_\ s, \mathsf{pack}\ \_\ s') \in S$ to conclude there is a QPER R such that $(s, s') \in [\![\Gamma, \alpha : \kappa \vdash X : *]\!]\ (\gamma, (\_, R)/\alpha)$. However, the QPER-join adds elements that are not in the union, so this does not immediately follow. Instead, we do induction on the join to show that there is some $(\mathsf{pack}\ \_\ t, \mathsf{pack}\ \_\ t') \sim (\mathsf{pack}\ \_\ s, \mathsf{pack}\ \_\ s')$ such that $(t, t') \in [\![\Gamma, \alpha : \kappa \vdash X : *]\!]\ (\gamma, (\_, R)/\alpha)$. Then, by the eta-rule for existentials, and transitivity of $\sim_{[\![\Gamma \vdash \exists \alpha : \kappa.\ X : *]\!]\ \gamma}$, we can conclude that $(e, e') \sim (\mathsf{pack}\ \_\ t, \mathsf{pack}\ \_\ t')$. ◀

That is, the two terms that witness the relation are not necessarily the *exact* terms that e and e' evaluate to, but rather are equivalent to them. (The same caveat holds for the existential equality principle in Plotkin-Abadi logic.) Because of this issue, we cannot give a direct realizer internalizing this reasoning principle in our type theory, as we need to extend the type theory with a form of proof-irrelevance first: knowing two existential packages are equal does not tell us which relation witnesses that equality! This we leave for future work.

However, we can still of course add equality axioms for particular instances of representation independence. For example, consider the following two existential packages:

$$\begin{array}{lcl} X & \triangleq & \exists \alpha : *.\ \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool}) \\ M : X & = & \mathsf{pack}\, \mathbb{N}\ (z, s, (\lambda n.\ n\ \mathsf{bool}\ \mathsf{true}\ (\lambda k.\ \mathsf{false}))) \\ O : X & = & \mathsf{pack}\, \mathsf{bool}\ (\mathsf{true}, (\lambda b.\ \mathsf{false}), (\lambda b.\ b)) \end{array}$$

This package exports a seed value, an operation on it, and a test that says whether the argument is the seed or not. Since M and O behave the same, we can relate them at existential type, and add $\mathsf{refl} : M =_X N$ as an axiom to our system.

5.5 Quotient Types

While not an application of parametricity in the sense of theorems for free [26], we can also show the realizability of quotient types [13] in our semantics. Quotient types, give a way to define new types by taking an existing type, and quotienting it by an equivalence relation.

To do this, we first define the auxiliary predicate Eq_X, which formalizes the notion of an equivalence relation. They are relations $R : X \to X \to *$, satisfying:

$$\begin{array}{lcl} \mathrm{Eq}_X(R) & \triangleq & \Pi x : X.\ R\ x\ x \times \\ & & \Pi x : X, y : X.\ R\ x\ y \leftrightarrow R\ y\ x \times \\ & & \Pi x : X, y : X, z : X.\ R\ x\ y \to R\ y\ z \to R\ x\ z \end{array}$$

Next, we can show the realizability of the following datatype:

$$\begin{array}{lcl} X/R & \triangleq & \exists \beta : *, \\ & & \Sigma \mathit{inj} : X \to \beta. \\ & & \Sigma \mathit{app} : \Pi \gamma : *.\ \Pi f : X \to \gamma. \\ & & \qquad (\Pi a : X, a' : X.\ R\ a\ a' \to f\ a =_\gamma f\ a') \to (\beta \to \gamma). \\ & & \Pi a : X, a' : X.\ R\ a\ a' \to \mathit{inj}(a) =_\beta \mathit{inj}(a') \times \\ & & \Pi \gamma.\ \Pi f, pf, x.\ \mathit{app}\ \gamma\ f\ pf\ (\mathit{inj}\ x) =_\gamma f\ x \end{array}$$

What we are doing is defining an existential type, such that if X is a type and R is an equivalence relation on it, we return a new type β and two operations *inj* and *app*.

The *inj* is the injection into the quotient type. It takes an X, and returns a β, with the property that if a and a' are related by R, then $\mathit{inj}\ a = \mathit{inj}\ a'$. The *app* function then lifts any function f from $X \to \gamma$ into one on $\beta \to \gamma$, provided that f respects the equivalence relation R. The last two lines give the equational theory of the quotient type. First, if a and a' are related by R, then $\mathit{inj}\ a = \mathit{inj}\ a'$. Second, if we lift a function f to operate on quotients, and we pass it the argument $\mathit{inj}\ x$, then the application of the lifted function should equal $f\ x$.

Proof. (Sketch) The proof of the soundness of the axiom proceeds quite directly. First, we define the following relation:

$$S = \{(e_1, e_2') \mid \exists e_1', e_2, \bar{q}.\ \ (e_1, e_1') \in [\![X]\!] \ \wedge\ (e_2, e_2') \in [\![X]\!] \ \wedge\ \bar{q} \in [\![R]\!]\ (e_1, e_1')\ (e_2, e_2')\}$$

As an abuse of notation, we suppress most of the context and environment arguments from the definition. By making use of the fact that we have a proof of $\mathrm{Eq}_X(R)$, we can show that

S is a QPER, and then use it as our witness for showing the existential type is inhabited. We can define *inj* and *app* as:

- $\mathsf{inj} = \lambda x : X.\ x$
- $\mathsf{app} = \lambda\gamma : *.\ \lambda f : X \to \gamma, pf : \ldots, x : X.\ f\ x$

and give two dummy realizers for the proofs:

- $\mathsf{equiv} = \lambda a, a', r.\ \mathsf{refl}$
- $\mathsf{appok} = \lambda\gamma.\ \lambda f, pf, x.\ \mathsf{refl}$

With these, we can then show the realizability of the term:

$$\mathsf{pack}\ X\ \mathsf{pair}\ \mathsf{inj}\ (\mathsf{pair}\ \mathsf{app}\ (\mathsf{pair}\ \mathsf{equiv}\ \mathsf{appok}))$$

paired with itself at the witness relation S. Note that this term is not well-typed in the syntactic system, but that it does inhabit the appropriate semantic type. In terms of the operational semantics of the underlying realizers, quotienting is a no-op: no representation changes are needed to protect the quotient type's invariant: data abstraction is enough. ◀

6 Discussion and Related Work

6.1 Quasi-PERs

The earliest use of quasi-PERs to to model data abstraction we have found is by Tennent and Takeyama [24], who were studying program equivalences in the context of data refinement. In this e-mail, they sketched a logical relations model of the simply-typed lambda calculus in terms of quasi-PERs (which they called "zigzag-complete relations"). The term "quasi-PER" was coined by Hofmann [14], who gave a logical relations model of a simply-typed functional language, augmented with first-order state (*i.e.,* references to integers). Though this language had no polymorphism, it did have effect annotations, and so Hofmann needed to prove representation independence to show the equivalence of programs with possibly-differing sets of effects (*i.e.,* "effect masking" [17]). Our work greatly extends the reach of quasi-PERs to support polymorphism, higher kinds, and type dependency, showing that the simple idea of quasi-PERs scales up even to a wide array of type-theoretic features.

Hutton and Voermans [16] studied a relational version of Squiggol [8], in which the category of sets and relations was replaced with the category of PERs and saturated relations In this setting, they observed that the functional relations were precisely the difunctionals. Many useful properties of quasi-PERs are worked out in this paper, even though their application of them is rather different from our own.

6.2 Semantic Models for Parametricity

The standard approach to building parametric models is to begin with a non-parametric model of the language, and then give a second interpretation of types as relations over the non-parametric semantic types. This approach is used in Bainbridge *et al.* [4], and an abstract characterization of it was given by Rosolini [22] using categories of *reflexive graphs*, which both Dunphy and Reddy [12] and Birkedal *et al.* [9] have developed further.

Our model does not immediately fit into this framework, since we give a relational semantics directly, *without* first building a non-parametric model. This represents a considerable simplification of the metatheory: we only need one semantics, rather than two.

Both Vytiniotis and Weirich [25] and Atkey [3] give parametric models of F_ω, which is equivalent to the calculus of constructions minus dependency. The work in [25] also gives

a term model, and as such they need to prove a coherence theorem for the interpretation of kinds. In contrast, Atkey [3] does not need to prove such a coherence theorem, since he defines his relation over extensional semantic objects. He does, however, need to prove the identity extension lemma, to connect his base semantics with his relational semantics.

The way we set things up means our proofs are overall a bit easier than either of these two approaches (modulo the additional overhead imposed by type dependency). In [25], great care is taken to ensure that their relations only mention well-formed type constructors. We do not bother maintaining this invariant: since the operational semantics never examines a type argument, there is no need for the model to worry if a type argument is well formed or not. By moving to a realizability-based view, we also make it possible to add *realizable axioms*: we can add any axioms we want, as long as those axioms have (possibly syntactically ill-typed) lambda terms as realizers for their computational behavior.

Even stating the identity extension property for higher kinds requires equipping each kind with a distinguished notion of identity relation, to connect the base and relational interpretations. We do not need to do this, since we only have a relational semantics. We still need to ensure that our interpretation of higher kinds respects the equalities (quasi-PER) on types, but found this easier to work with than identity extension, since we only need to consider the base kind.

6.3 Internalizations of Parametricity

In recent work, Bernardy *et al.* [6] demonstrate how to generalize Reynolds' relational interpretation to systems of dependent types, and show that for sufficiently rich type theories, the image of the relational interpretation lies within the original type theory. This gives a syntax-directed embedding of a parametric interpretation of type theory into itself.

The translational approach is wholly syntactic, as opposed to our more semantic approach. One benefit of their method is that it yields concrete proof terms for parametricity properties. The two principal limitations of the translational approach are that (1) parametricity properties only apply to closed expressions, and (2) there is no way to use parametricity to internalize program equivalences as equalities.

In [7], Bernardy and Moulin relax the first restriction by extending the syntax of type theory with *operators* to represent appeals to parametricity. Though the syntactic modifications they make to type theory are quite complex, the fundamental idea is quite simple: they are using indices to indicate the "color" of different subterms [5], and parametricity lets them show that different colors do not interfere with one another. However, the second restriction remains, and intentionally so. They support inductive definitions in the style of Coq and Agda, which permits internalizing the conversion relation by defining the Martin-Löf identity type as an inductive type. This inherently limits what equality can contain: the Church booleans cannot be shown to be equal to true or false, unless the conversion relation contains it. This approach might be described as "Strachey-style" [23], where the *uniformity* of parametric computations allows deriving powerful and elegant erasure properties.

The strengths and weaknesses of our approach are reversed. We do not give full proof terms, due to our use of equality reflection, but we can support parametricity arguments for open terms, and can internalize parametric program equivalences as equalities. Our approach can be viewed as "Reynolds-style" parametricity, where the emphasis is on the *relational* character of parametricity, leading to a focus on representation independence and eta-laws. A natural question is whether it is possible to combine the strengths of these two approaches, and gain the decidability advantages of their approach while retaining the simple interface to parametricity we can support.

References

1 S. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4), 2006.

2 T. Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

3 R. Atkey. Relational parametricity for higher kinds. In *Computer Science Logic*, 2012.

4 E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theor. Comput. Sci.*, 70(1), 1990.

5 J.-P. Bernardy. Type theory in color. Preprint, 2012.

6 J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *ICFP 2010*. ACM, 2010.

7 J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *LICS*. IEEE Computer Society, 2012.

8 R. Bird and O. de Moor. *Algebra of Programming*. International Series in Computing Science, Vol. 100. Prentice Hall, 1997.

9 L. Birkedal, R. E. Møgelberg, and R. L. Petersen. Domain-theoretical models of parametric polymorphism. *Theor. Comput. Sci.*, 388(1-3), 2007.

10 T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3), Feb. 1988.

11 D. Doel. Proving induction principles via free theorems from parametricity. Agda code at `http://code.haskell.org/~dolio/ agda-share/ParamInduction.html`, 4 April 2012.

12 B. P. Dunphy and U. S. Reddy. Parametric limits. In *LICS*. IEEE Computer Society, 2004.

13 M. Hofmann. A simple model for quotient types. In *Typed Lambda Calculi and Applications*. 1995.

14 M. Hofmann. Correctness of effect-based program transformations. In O. Grumberg, T. Nipkow, and C. Pfaller, editors, *Formal Logical Methods for System Security and Correctness*, volume 14. IOS Press, 2008.

15 M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five Years of Constructive Type Theory*. Oxford University Press, 1998.

16 G. Hutton and E. Voermans. Making Functionality More General. In *Glasgow Workshop on Functional Programming*, Skye, Scotland, 1992.

17 J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL 1988*. ACM, 1988.

18 P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis Naples, Italy, 1984.

19 J. McKinna. Why dependent types matter. In *POPL*, 2006.

20 G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA*, 1993.

21 J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

22 E. P. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *LICS*. IEEE Computer Society, 1994.

23 C. Strachey. Fundamental concepts in programming languages. *HOSC*, 13(1), 2000 (original lectures in 1967).

24 R. Tennent and M. Takeyama. What is a data refinement relation? E-mail to `data-refinement@etl.gp.jp`, January 1996.

25 D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Functional Programming*, 20(2), March 2010.

26 P. Wadler. Theorems for free! In *FPCA 1989*. ACM, 1989.