A Relationally Parametric Model of the Calculus of Constructions

Neelakantan R. Krishnaswami Derek Dreyer
Max Planck Institute for Software Systems (MPI-SWS)
{neelk,dreyer}@mpi-sws.org

Abstract

In this paper, we give the first relationally parametric model of the (extensional) calculus of constructions. Our model remains as simple as traditional PER models of dependent types, but unlike them, our model additionally permits relating terms at *different* implementation types. Using this model, we can validate the soundness of quotient types, as well as derive strong equality axioms for Church-encoded data, such as the eta-law for strong dependent pair types.

1. Introduction

Reynolds [26] introduced the concept of *relational parametricity* with a fable about data abstraction: professors Bessel and Descartes, each teaching a class on complex variables, defined complex numbers differently in the first lecture, in polar and cartesian terms, respectively. 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 the underlying representation of the complex numbers.

He formalized this idea by giving a relational semantics for System F, in which each type denoted not just a set of well-formed terms, but a relation between them. Then, the fact that client programs were insensitive to a specific choice of implementation could be formalized in terms of well-typed programs taking related inputs to related results. Since the two constructions of the complex numbers share the same interface and are related, any program which uses complex numbers must return equivalent answers, regardless of the choice of implementation. Hence parametricity gives a way of modeling a general notion of program equivalence, significantly stronger than basic notions of set-theoretic equality.

Subsequently, Plotkin and Abadi [25] showed how to build a program logic for proving the equivalence of programs in System F using parametricity properties. Plotkin-Abadi logic is a system in the LCF tradition, where an external logic is used to reason about the behavior 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 [24]. Dependent type systems allow types to mention program terms, which thereby allows programmers to define types stating strong invariants about the behavior of a program. The ability to put strong preconditions on functions means that we can support operations that might be unsafe (e.g., unchecked array access) without compromising the safety of the programming language. Essentially, with such expressive type systems, type-correctness then implies satisfaction of specifications.

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 closure of β-reduction for lambda-terms. Even more generous approaches, such as denotational models based on locally cartesian closed categories [28] or extensional type theories [1, 23], still equip 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. [8], in which they show by translation that the *syntax* of type theory is wholly compatible with parametricity — every well-typed term in the calculus of constructions respects the relational interpretation, but there is no way to internalize it.

As a result, it is possible to *define* many types such as existentials, coproducts, and dependent pairs, but it is not possible to prove that they satisfy the expected equational properties (e.g., η-rules and commuting conversions). This, despite the fact that they *do* satisfy those properties! 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 calculus of constructions [12]. Our model permits us to support a *parametric identity type*: our model of the identity type of respects data abstraction, and consequently we can show the soundness of equality axioms whose correctness relies upon parametricity.

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 only proves that the relation is a reflexive congruence for well-typed terms (the "fundamental property"). The relation is not neccessarily symmetric or transitive, both of which are needed to use the relation to interpret equality for programs. (The failure of symmetry and transitivity is what prevented our earlier work [20] from giving a useful *judgement* for deriving equalities, as opposed to simply giving semantic side-conditions appealing to the model.)

A common approach to gaining symmetry and transitivity is to require the relations interpreting types to be partial equivalence relations (PERs). Since PERs are symmetric and transitive, this con-

dition automatically ensures equivalence is symmetric and transitive. 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.

Our key innovation is to generalize from modeling types as PERs to modeling them as *quasi-PERs* (a.k.a. *difunctional relations*, or *zig-zag complete relations*), which generalize partial equivalence relations to the asymmetric case. Using quasi-PERs, we give a relational model permitting *both* the comparison of terms with differing representations, *and* proofs of the symmetry and transitivity of equality.

To illustrate this point, we show how to state parametricity axioms for some types familiar from System F such as units and existentials. Making more use of 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.

2. Type System 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.

In Figure 1, we give the syntactic categories of our type system. We present our system with distinct syntactic categories for kinds (ranged over by metavariables κ), types (ranged over by metavariables X, Y, A, B, 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 and B for type constructors of arbitrary kind, and X and Y for type constructors of base kind.

The kinds include the base kind of types $*$, the kind of term-indexed types $\Pi x : X.\ \kappa$, and the kind of type-indexed types $\Pi\alpha : \kappa.\ \kappa'$. Our base type constructors include universal quantification $\forall\alpha : \kappa.\ X$, the dependent function space $\Pi x : X.\ Y$, and the identity type $e_1 =_X e_2$. On top of this, we permit abstracting over both term variables $\lambda x : X.\ A$ and type variables $\lambda\alpha : \kappa.\ A$, with corresponding applications $A\ e$ and $A\ B$. Of course, we can also refer to type variables α in type expressions.

The syntax of terms is also explicitly typed, with both explicitly-typed term $\lambda x : X.\ e$ and type $\Lambda\alpha : \kappa.\ e$ lambda-abstractions, and corresponding applications $e\ e'$ and $e\ A$. The witness for equality proofs is the reflexivity term refl. There is no elimination form for this type, since we make use of the equality reflection principle [23], and therefore do not need an explicit eliminator for equality. We identify a subclass of terms v as values, which we take to be the lambda-terms $\lambda x : X.\ e$ and $\Lambda\alpha : \kappa.\ e$, as well as the equality proof term refl.

As an aside, we present a system with distinct syntactic levels rather than as a pure type system [6]. Having different syntactic categories and judgements simplifies some theorem statements, but comes at the price of doubling the number of substitution lemmas, since we have to prove substitution properties once each for terms and types.

In Figure 2, we catalog the judgements we use in our system.

We have the judgement $\Gamma\ \mathsf{ok}$, which asserts that a context (consisting of term and type variables) is well-formed, and the judgement $\Gamma \vdash \kappa : \mathsf{kind}$, which describes when a kind is well-formed. Both of these judgements are given in Figure 3. Note that the base case of the well-kinding judgement — $\Gamma \vdash * : \mathsf{kind}$ does not require that the context be well-formed.

This illustrates a general principle in our choice of typing rules. We avoid making a context well-formedness requirement part of the derivations of our system. Instead, we state the well-formedness condition as a precondition to the theorems of our metatheory, and add sufficient premises to ensure that the context well-formedness condition can be derived as needed. For example, the rule for pi-kinds $\Gamma \vdash x : X : \mathsf{kind}$ has a premise that $\Gamma \vdash X : *$, which lets us derive the well-formedness of the extended context $\Gamma, x : X$. One further convention that we follow is that our rules have implicit validity premises – if we have a rule ending in $\Gamma \vdash A : \kappa$, then we implicitly require in the premise that $\Gamma \vdash \kappa : \mathsf{kind}$. This simplifies the soundness proof, but since these premises add clutter to the rules we omit them in the display.

In Figure 4, we give the well-kinding judgement $\Gamma \vdash A : \kappa$, which asserts that the type constructor A has the kind κ. The rules here are unsurprising: we have type variables α, abstractions over terms $\lambda x : X.\ A$ and types $\lambda\alpha : \kappa.\ A$, as well as the corresponding applications $A\ e$ and $A\ B$. We also have a few basic rules for forming types (of kind $*$) — polymorphic functions $\forall\alpha : \kappa.\ X$, dependent functions $\Pi x : X.\ Y$, and the identity type $e_1 =_X e_2$. Finally, we have a conversion rule which says that if A has the kind κ, and $\kappa \equiv \kappa'$, then A also has the kind κ'.

In Figure 5 we give the typing rules for expressions. Variables are looked up in the environment, and we have the expected rules for abstractions and applications over types and terms. There is also a conversion rule for typing terms, and finally we have the rule for equality, which asserts that if e_1 and e_2 are convertible, then refl is a proof of inhabitation of the identity type $e_1 =_X e_2$.

The conversion rules for kinds and types are given in Figure 6. Equivalence of kinds is basically structural – if two kinds have the same shape, and equal type and term components, then they are equal. We model this by giving a substitution principle that says if we substitute equal terms or types into a kind, then the two resulting kinds are equal. Then, we close up under reflexivity, symmetry, and transitivity.

The equality judgement for type constructors is similar. As with kinds, we express the fact that equality is a congruence by giving rules which assert that substituting equal types and terms into a type yields equal results, and we also close up under reflexivity, symmetry, and transitivity. Since kinds have their own notion of equality, we also say that an equality proved at one kind is valid at any equal kind. Then, we add the β and η-rules of the lambda-calculus for the type and term abstractions in the language.

Similarly, we give the equality judgement for expressions in Figure 7. As with kinds and types, we use substitution of equal terms to express that the relation is a congruence, have reflexivity, transitivity, and symmetry rules, and a rule asserting that an equality at one type is also an equality at any equal type. Then, we have the β and η rules for type and term abstractions.

Identity types support a proof-irrelevance principle, called Axiom K [17], which asserts all proofs of equalities are equal. Furthermore, term equality also contains the equality reflection rule. If $\Gamma \vdash e_p : e =_X e'$, then $\Gamma \vdash e \equiv e' : X$. Note that this rule makes typechecking undecidable, since a well-typing derivation may need to invent equality proofs out of thin air.

In Figure 8, we give the operational semantics of our programming language. This is a standard big-step, call-by-name semantics, with no surprises to it. The reason that we give an operational semantics, rather than (say) a β-convertibility relation, is that full-β does not have to terminate. Since we support the equality reflection principle, we can prove the well-typedness of the Y-combinator in open contexts – so the arbitrary β-reduction of well-typed open terms is not necessarily guaranteed to terminate. However, since the system is (as we will prove) consistent, this means that all *closed* terms do reduce to values. This problem, and our approach to resolving it, are both quite familiar from Nuprl.

Another point worth noting is that our definitional equality is just the $\beta\eta$-theory of the lambda calculus (plus axiom K for identity

$$
\begin{array}{llll}
\kappa & ::= & * \mid \Pi x : X.\ \kappa \mid \Pi \alpha : \kappa.\ \kappa' & \text{Kinds} \\
X, A & ::= & \forall \alpha : \kappa.\ X \mid \Pi x : X.\ Y \mid e =_X e' & \text{Types} \\
 & \mid & \lambda x : X.\ A \mid A\ e \mid & \\
 & \mid & \lambda \alpha : \kappa.\ A \mid A\ B \mid \alpha & \\
e & ::= & x \mid \lambda x : X.\ e \mid \mid e\ e & \text{Terms} \\
 & \mid & \Lambda \alpha : \kappa.\ e \mid e\ A \mid \mathsf{refl} & \\
v & ::= & \lambda x : X.\ e \mid \Lambda \alpha : \kappa.\ e \mid \mathsf{refl} & \text{Values} \\
\Gamma & ::= & \cdot \mid \Gamma, x : X \mid \Gamma, \alpha : \kappa & \text{Contexts}
\end{array}
$$

Figure 1. Syntax

$\Gamma\ \mathsf{ok}$	Context Well-formedness
$\Gamma \vdash \kappa : \mathsf{kind}$	Kind Well-formedness
$\Gamma \vdash A : \kappa$	Well-kinding for 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 Kinds
$\Gamma \vdash e \equiv e' : X$	Definitional Equality for Kinds
$e \Downarrow v$	Operational Semantics

Figure 2. Summary of Judgments

$$\boxed{\Gamma\ \mathsf{ok}}$$

$$\frac{}{\cdot\ \mathsf{ok}}$$

$$\frac{\Gamma\ \mathsf{ok} \qquad \Gamma \vdash X : *}{\Gamma, x : X\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \qquad \Gamma \vdash \kappa : \mathsf{kind}}{\Gamma, \alpha : \kappa\ \mathsf{ok}}$$

$$\boxed{\Gamma \vdash \kappa : \mathsf{kind}}$$

$$\frac{}{\Gamma \vdash * : \mathsf{kind}} \qquad \frac{\Gamma \vdash X : * \qquad \Gamma, x : X \vdash \kappa : \mathsf{kind}}{\Gamma \vdash \Pi x : X.\ \kappa : \mathsf{kind}}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \kappa' : \mathsf{kind}}{\Gamma \vdash \Pi \alpha : \kappa.\ \kappa' : \mathsf{kind}}$$

Figure 3. Context and Kind Well-formedness

types). We have not yet included any parametricity properties in our rules for equality. This choice is made for expository purposes. In this way, we keep the base system relatively conventional, and can consolidate all of the parametricity axioms we use into our examples in Section 4.

3. Model

In this section, we describe the model construction we use to interpret the calculus of constructions. First, we describe what quasi-PERs are, and then we describe how we interpret each of the judgements of our type theory – contexts, kinds, types, terms, and equalities.

$$\boxed{\Gamma \vdash A : \kappa}$$

$$\frac{\Gamma \vdash \kappa : \mathsf{kind} \qquad \Gamma, \alpha : \kappa \vdash Y : *}{\Gamma \vdash \forall \alpha : \kappa.\ Y : *}$$

$$\frac{\Gamma \vdash X : * \qquad \Gamma, x : X \vdash Y : *}{\Gamma \vdash \Pi x : X.\ Y : *} \qquad \frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : X}{\Gamma \vdash e =_X e' : *}$$

$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash X : * \qquad \Gamma, x : X \vdash A : \kappa}{\Gamma \vdash \lambda x : X.\ A : \Pi x : X.\ \kappa}$$

$$\frac{\Gamma \vdash \kappa : \mathsf{kind} \qquad \Gamma, \alpha : \kappa \vdash A : \kappa'}{\Gamma \vdash \lambda \alpha : \kappa.\ A : \Pi \alpha : \kappa.\ \kappa'}$$

$$\frac{\Gamma \vdash A : \Pi x : X.\ \kappa \qquad \Gamma \vdash e : X}{\Gamma \vdash A\ e : [e/x]\kappa}$$

$$\frac{\Gamma \vdash A : \Pi \alpha : \kappa.\ \kappa' \qquad \Gamma \vdash A' : \kappa}{\Gamma \vdash A\ A' : [A'/\alpha]\kappa'}$$

$$\frac{\Gamma \vdash A : \kappa' \qquad \Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}}{\Gamma \vdash A : \kappa}$$

Figure 4. Kinding for Type Constructors

$$\boxed{\Gamma \vdash e : X}$$

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X} \qquad \frac{\Gamma \vdash e : Y \qquad \Gamma \vdash X \equiv Y : *}{\Gamma \vdash e : X}$$

$$\frac{\Gamma \vdash \kappa : \mathsf{kind} \qquad \Gamma, \alpha : \kappa \vdash e : Y}{\Gamma \vdash \Lambda \alpha : \kappa.\ e : \forall \alpha : \kappa.\ Y}$$

$$\frac{\Gamma \vdash e : \forall \alpha : \kappa.\ Y \qquad \Gamma \vdash A : \kappa}{\Gamma \vdash e\ A : [A/\alpha]Y}$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x : X.\ e : \Pi x : X.\ Y}$$

$$\frac{\Gamma \vdash e : \Pi x : X.\ Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e\ e' : [e'/x]Y}$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : X}{\Gamma \vdash \mathsf{refl} : e_1 =_X e_2}$$

Figure 5. Expression Typing

$$\boxed{\Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash e \equiv e' : X \qquad \Gamma, x : X \vdash \kappa : \mathsf{kind}}{\Gamma \vdash [e/x]\kappa \equiv [e'/x]\kappa : \mathsf{kind}}$$

$$\frac{\Gamma \vdash A \equiv A' : \kappa \qquad \Gamma, \alpha : \kappa \vdash \kappa' : \mathsf{kind}}{\Gamma \vdash [A/\alpha]\kappa' \equiv [A'/\alpha]\kappa' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash \kappa : \mathsf{kind}}{\Gamma \vdash \kappa \equiv \kappa : \mathsf{kind}} \qquad \frac{\Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}}{\Gamma \vdash \kappa' \equiv \kappa : \mathsf{kind}}$$

$$\frac{\Gamma \vdash \kappa_1 \equiv \kappa_2 : \mathsf{kind} \qquad \Gamma \vdash \kappa_2 \equiv \kappa_3 : \mathsf{kind}}{\Gamma \vdash \kappa_1 \equiv \kappa_3 : \mathsf{kind}}$$

$$\boxed{\Gamma \vdash A \equiv A' : \kappa}$$

$$\frac{\Gamma \vdash e \equiv e' : X \qquad \Gamma, x : X \vdash A : \kappa}{\Gamma \vdash [e/x]A \equiv [e'/x]A : [e/x]\kappa}$$

$$\frac{\Gamma \vdash A \equiv A' : \kappa \qquad \Gamma, \alpha : \kappa \vdash B : \kappa'}{\Gamma \vdash [A/\alpha]B \equiv [A'/\alpha]B : [A/\alpha]\kappa'}$$

$$\frac{\Gamma \vdash A \equiv A' : \kappa' \qquad \Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}}{\Gamma \vdash A \equiv A' : \kappa}$$

$$\frac{\Gamma \vdash A : \kappa}{\Gamma \vdash A \equiv A : \kappa} \qquad \frac{\Gamma \vdash A \equiv A' : \kappa}{\Gamma \vdash A' \equiv A : \kappa}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 : \kappa \qquad \Gamma \vdash A_2 \equiv A_3 : \kappa}{\Gamma \vdash A_1 \equiv A_3 : \kappa}$$

$$\frac{\Gamma \vdash \lambda x : X.\ A : \Pi x : X.\ \kappa \qquad \Gamma \vdash e : X}{\Gamma \vdash (\lambda x : X.\ A)\ e \equiv [e/x]A : [e/x]\kappa}$$

$$\frac{\Gamma \vdash \lambda \alpha : \kappa.\ A : \Pi \alpha : \kappa.\ \kappa' \qquad \Gamma \vdash A' : \kappa'}{\Gamma \vdash (\lambda \alpha : \kappa.\ A)\ A' \equiv [A'/\alpha]A : [A'/\alpha]\kappa'}$$

$$\frac{\begin{array}{c}\Gamma, x : X \vdash A\ x \equiv A'\ x : \kappa \\ \Gamma \vdash A : \Pi x : X.\ \kappa \qquad \Gamma \vdash A' : \Pi x : X.\ \kappa\end{array}}{\Gamma \vdash A \equiv A' : \Pi x : X.\ \kappa}$$

$$\frac{\begin{array}{c}\Gamma, \alpha : \kappa \vdash A\ \alpha \equiv A'\ \alpha : \kappa' \\ \Gamma \vdash A : \Pi \alpha : \kappa.\ \kappa' \qquad \Gamma \vdash A' : \Pi \alpha : \kappa.\ \kappa'\end{array}}{\Gamma \vdash A \equiv A' : \Pi \alpha : \kappa.\ \kappa'}$$

Figure 6. Kind and Type Equality

$$\boxed{\Gamma \vdash e_1 \equiv e_2 : X}$$

$$\frac{\Gamma \vdash e_p : e =_X e'}{\Gamma \vdash e \equiv e' : X} \qquad \frac{\Gamma \vdash e \equiv e' : Y \qquad \Gamma \vdash X \equiv Y : *}{\Gamma \vdash e \equiv e' : X}$$

$$\frac{\Gamma \vdash e_0 \equiv e_0' : Y \qquad \Gamma, x : Y \vdash e : X}{\Gamma \vdash [e_0/x]e \equiv [e_0'/x]e : [e_0/x]X}$$

$$\frac{\Gamma \vdash A \equiv A' : \kappa \qquad \Gamma, \alpha : \kappa \vdash e : X}{\Gamma \vdash [A/\alpha]e \equiv [A'/\alpha]e : [A/\alpha]X}$$

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash e \equiv e : X} \qquad \frac{\Gamma \vdash e \equiv e' : X}{\Gamma \vdash e' \equiv e : X}$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : X \qquad \Gamma \vdash e_2 \equiv e_3 : X}{\Gamma \vdash e_1 \equiv e_3 : X}$$

$$\frac{\Gamma \vdash \Lambda \alpha : \kappa.\ e : \forall \alpha : \kappa.\ X \qquad \Gamma \vdash A : \kappa}{\Gamma \vdash (\Lambda \alpha : \kappa.\ e)A \equiv [A/\alpha]e : [A/\alpha]X}$$

$$\frac{\begin{array}{c}\Gamma, \alpha : \kappa \vdash e\ \alpha \equiv e'\ \alpha : Y \\ \Gamma \vdash e : \forall \alpha : \kappa.\ Y \qquad \Gamma \vdash e' : \forall \alpha : \kappa.\ Y\end{array}}{\Gamma \vdash e \equiv e' : \forall \alpha : \kappa.\ Y}$$

$$\frac{\Gamma \vdash \lambda x : X.\ e : \Pi x : X.\ Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash (\lambda x : X.\ e)\ e' \equiv [e'/x]e : [e'/x]Y}$$

$$\frac{\begin{array}{c}\Gamma, x : X \vdash e\ x \equiv e'\ x : Y \\ \Gamma \vdash e : \Pi x : X.\ Y \qquad \Gamma \vdash e' : \Pi x : X.\ Y\end{array}}{\Gamma \vdash e \equiv e' : \Pi x : X.\ Y}$$

$$\frac{\Gamma \vdash e : e_1 =_X e_2 \qquad \Gamma \vdash e' : e_1 =_X e_2}{\Gamma \vdash e \equiv e' : e_1 =_X e_2}$$

Figure 7. Expression Equality

$$\boxed{e \Downarrow v}$$

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow \lambda x : X.\ e'' \qquad [e'/x]e'' \Downarrow v}{e\ e' \Downarrow v}$$

$$\frac{e \Downarrow \Lambda \alpha : \kappa.\ e' \qquad [X/\alpha]e' \Downarrow v}{e\ X \Downarrow v}$$

Figure 8. Operational Semantics

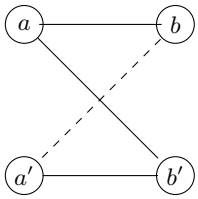


Figure 9. Quasi-PER

3.1 Quasi-PERs

A *quasi-PER* (or quasi-PER, also known as a "difunctional relation", or "zig-zag complete relation"), is a relaxation of the concept of a partial equivalence relation to the asymmetric case. Formally, they are defined as follows:

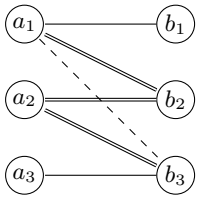
Definition 1. (Quasi-PER) Given sets A and B, a *quasi-PER* $R \subseteq A \times B$ is a relation such that for all $a, a' \in A$ and $b, b' \in B$, if $(a, b) \in R$ and $(a', b') \in R$ and $(a, b') \in R$, then $(a', b) \in R$.

In Figure 9, we give a diagram of the quasi-PER condition, which should illustrate why they are also called "zig-zag complete relations" — if there is a "zig" between two pairs of related points, then there must also be a "zag".

For our purposes, quasi-PERs are interesting because their asymmetry lets us prove representation independence results (i.e., we can relate different representations of a datatype), without losing the possibility of using the logical relation to *define* equality of datatypes.

To understand why, suppose that we have a type X, which is interpreted by a relation $R \subseteq A \times B$. Furthermore, suppose that we have three terms of type X, e_1, e_2 and e_3, with the property that $e_1 = e_2$ and $e_2 = e_3$ according to the equational theory of the language.

To show transitivity, we will need the following two properties. First, we will need a version of the fundamental theorem of logical relations, to tell us that each e_i is interpreted by $(a_i, b_i) \in R$. Secondly, we will need the soundness of the equality rules to tell us that $e_1 = e_2$ means $(a_1, b_2) \in R$ and that $e_2 = e_3$ means that $(a_2, b_3) \in R$. Then, the fact that R is a quasi-PER implies that (a_1, b_3) is also in R. Below, we illustrate how the zig-zag condition implies $(a_1, b_3) \in R$, by doubling the three lines that we use to reach the conclusion.



Just as with ordinary PERs, quasi-PERs are closed under arbitrary intersections, but also like PERs they are not closed under unions. However, since the intersection equips quasi-PERs with a complete semilattice structure, we can define the join of a set of quasi-PERs as the least quasi-PER containing the union. We can also define the join directly, as the "zig-zag" closure, where we add the pairs necessary to ensure that the zig-zag condition holds. The construction closely resembles the join on partial equivalence relations, and just as with PERs, existential or union types defined using the join only support a weak (unpack-style) elimination form, rather than a projective elimination.

Notation If Q is a quasi-PER, then we will write $\bar{e} \in Q$ to denote a pair $(e, e') \in Q$. For $(a, b) \in Q$ and $(a', b') \in Q$, we will write $(a, b) \sim_Q (a', b')$ if $(a, b') \in Q$ and/or $(a', b) \in Q$. (We say "and/or" because "and" is equivalent to "or" here thanks to zig-zag completeness). We will also suppress the subscript and write $\overline{e_1} \sim \overline{e_2}$ when Q is obvious from context.

3.2 The Semantic Interpretation

We interpret relations with a set of mutually-recursive semantic interpretation functions, which we describe below. Given the high degree of mutual recursion, there is unfortunately no way to describe the interpretations without some degree of forward reference, though we have tried to minimize the number of forward references.

3.2.1 Contexts

The interpretation of the Γ ok judgement is the *set of grounding parallel substitutions* which satisfy it. We give the interpretation in Figure 10.

The interpretation of an empty context is just an empty substitution, and the interpretation of the context $\Gamma, x : X$ ok is an element of $[\![\Gamma \text{ ok}]\!]$, together with a pair $\bar{e}$ of closed terms from the interpretation of $\Gamma \vdash X : *$. The interpretation of the context $\Gamma, \alpha : \kappa$ ok is an element of $[\![\Gamma \text{ ok}]\!]$, together with a *triple* (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 types: 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 10, we also define a notion of equivalence $\gamma \sim_\Gamma \gamma'$ on contexts. This relation says that the relation part of type variables must be equal (ignoring the syntactic type constructors), and that pairs of terms $\overline{e_1}/x$ and $\overline{e_2}/x$ must lie in the same equivalence class of the relation. This definition does form a quasi-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" — note that the choice of quasi-PER to interpret types and kinds comes from the left-hand side. By construction, this choice will not matter, since all of our semantic functions will be invariant under this equivalence, but we cannot show that yet.

In Figure 11, we give some notations on contexts that we will use in the sequel. An element γ contains left- and right-bindings for each of the variables in it, and γ_1 is the left projection of the environment, and γ_2 is the right projection of the environment. We write $\overline{\gamma(e)}$ to indicate the pair of terms we get from the left and right substitutions of γ applied to e.

3.3 Kinds

We give the interpretation of derivations of the kind judgement in Figure 12. Since the judgement has a context Γ in it, our interpretation is relative to a context γ.

The interpretation of the base kind $\Gamma \vdash * : \mathsf{kind}$ is the set of quasi-PERs on closed values. The interpretation of the higher kind $\Gamma \vdash \Pi\alpha : \kappa.\ \kappa' : \mathsf{kind}$ is a currying of the interpretation $\Gamma, \alpha : \kappa \vdash \kappa' : \mathsf{kind}$. However, it is not the full function space: we restrict ourselves to the subset of functions which ignore the syntactic part of an argument triple $(\overline{X}, R)$.

Similarly, the interpretation of the higher kind $\Gamma \vdash \Pi x : X.\ \kappa' : \mathsf{kind}$ is a subset of the currying of the interpretation $\Gamma, x : X \vdash \kappa' : \mathsf{kind}$. However, in this case, we require that the type constructor return the same answer for all equivalent $\bar{e} \sim_X \overline{e'}$.

3.4 Type Constructors

In Figure 13, we give the interpretation of the type constructors of our language, as a function that takes a derivation and returns an element of the appropriate semantic kind. The first line of the

definition says that the interpretation of a derivation $\Gamma \vdash \alpha : \kappa$ proceeds by looking up α in the environment argument γ, and returning the relation component of the triple.

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 $\bar{e}$, and returns the interpretation of A in an extended environment, and the application $A\ e$ passes the substitution instance of $\overline{\gamma(e)}$ to the interpretation of A.

When an equality rule is used, we simply interpret the subderivation and return that as our answer. The presence of equalities is why we interpret full typing derivations. As a result, however, we will need to do a coherence proof on our semantic interpretations, though the fact that the equality rules do nothing interesting means the coherence proof is easy.

Next, we give the interpretations of types. The kinding interpretation requires that each type be a quasi-PER on closed values, and so we define each type as a relation on values. The interpretation of the function type $\Pi x : X.\ Y$ is the set of pairs of lambda-terms that take related pairs of arguments in X to related pairs of expressions in the expression relation for Y, in the context extended by the argument pair. This is basically the same as the usual rule for function types in logical relations, minimally adjusted to support dependency. Likewise, the interpretation of the polymorphic type $\forall\alpha : \kappa.\ X$ says that a pair of big lambdas is in the relation, if for every relation R in the kind κ, the bodies are related at the expression relation for X, in the environment augmented with R for α.

The interpretation for the identity type $e_1 =_X e_2$ is that the identity type equals $\{(\mathsf{refl}, \mathsf{refl})\}$ when e_1 and e_2 are related, and is the empty set otherwise. Observe that refl is a proof *only* when the equality holds, and that because the identity type is interpreted by a relation containing at most one pair, our model ensures it satisfies axiom K.

The Expression Relation While our interpretation of types defines a relation on *values*, our type theory deals in *expressions* — the operational semantics, the interpretation of contexts, and the equality judgement all replace variables with general expressions. To handle this, we introduce an auxilliary relation $\mathcal{E}\,[\![\Gamma \vdash X : *]\!]\ \ \gamma$, which defines a quasi-PER on closed expressions. This relation simply says that e and e' are related in the expression relation for X, just when they respectively reduce to a v and v' in the value relation of X.

3.5 Other Judgements

Since we are building a term model, we do not need to give an explicit interpretation of the expression typing or equality derivations. We will establish that we got these rules right as part of the proof of the soundness theorem, when we prove the fundamental lemma of logical relations and show that the syntactic equality judgement is sound with respect to the semantic expression relation.

As a result, at this point in the paper, we have not yet established that our definition is actually well-defined. We defer this proof to section 5. The reason is that the structure of a dependent type theory means that the well-definedness of our semantics is mutually inductive with the actual soundness property for the type theory. So before we state the soundness theorem, we will give examples to illustrate how the model works.

4. Examples of Parametricity

When proving the soundness of parametricity properties, our model is much like other parametric models. The only additional caveat we have to be bear in mind is that all our relations for types must be quasi-PERs. Fortunately, this is not a big limitation.

$$
\begin{array}{lcl}
(\langle\rangle)_1 & = & \langle\rangle \\
(\gamma, (e_1, e_2)\,/x)_1 & = & \gamma_1, e_1/x \\
(\gamma, ((A_1, A_2), R)/\alpha)_1 & = & \gamma_1, A_1/\alpha \\
 & & \\
(\langle\rangle)_2 & = & \langle\rangle \\
(\gamma, (e_1, e_2)\,/x)_2 & = & \gamma_2, e_2/x \\
(\gamma, ((A_1, A_2), R)/\alpha)_2 & = & \gamma_2, A_2/\alpha \\
 & & \\
\overline{\gamma(e)} & = & (\gamma_1(e), \gamma_2(e)) \\
\overline{\gamma(A)} & = & (\gamma_1(A), \gamma_2(A)) \\
\overline{\gamma(\kappa)} & = & (\gamma_1(\kappa), \gamma_2(\kappa))
\end{array}
$$

Figure 11. Notation

4.1 The Unit Type

It is traditional to illustrate parametricity with the easy cases of the unit type, and we are no exception. We can define the unit type $1 = \forall\alpha : *.\ \alpha \to \alpha$.

We begin by proving the soundness of the following lemma:

Lemma 1. *(All Units Equal the Identity) The following axiom is sound:*

$$\Pi p : 1.\ \forall\alpha : *.\ \Pi x : \alpha.\ p\ \alpha\ x =_\alpha x$$

Proof. We will go into a little detail into this proof, since it illustrates some of the particularities of working with quasi-PERs, in a simple setting. So, we will take $r = \lambda p.\ \Lambda\alpha : *.\ \lambda x : \alpha.$ refl as the realizer for this axiom, and show that $\bar{r}$ inhabits the type above.

To do so, we assume we have a $\bar{p} \in [\![\cdot \vdash 1 : *]\!]\ \langle\rangle$. Now we can just follow our noses for a bit. Assume we have $(\overline{A}, R) \in [\![\cdot \vdash * : \mathsf{kind}]\!]\ \langle\rangle$, and $(e, e') = \bar{e} \in \mathcal{E}\,[\![\alpha : * \vdash \alpha : *]\!]\ ((\overline{A}, R)/\alpha) \triangleq \hat{R}$. So this means we have (v, v') such that $e \Downarrow v$ and $e' \Downarrow v'$ and also $\bar{v} \in [\![\alpha : * \vdash \alpha : *]\!]\ ((\overline{A}, R)/\alpha)$. Now, to show refl inhabits the equality type, we need to show that $(p\ A\ e, e') \in \hat{R}$.

To do this, observe that we have $\bar{p} \in [\![\cdot \vdash 1 : *]\!]\ \langle\rangle$, and we are free to instantiate it at any relation we like. We will choose the relation $S = \{\bar{u} \mid \bar{v} \sim_R \bar{u}\}$. This small quasi-PER basically just picks out the equivalence class of R that $\bar{v}$ inhabits. But by parametricity, we know that $\overline{p\ A} \in \mathcal{E}\,[\![\alpha : * \vdash \alpha \to \alpha : *]\!]\ ((\overline{A}, S)/\alpha)$.

Let $\hat{S} = \mathcal{E}\,[\![\alpha : * \vdash \alpha : *]\!]\ ((\overline{A}, S)/\alpha)$. Because $\bar{v} \in S$, we have that $\bar{e} \in \hat{S}$. So we know that $\overline{p\ A\ e} \in \hat{S}$. But by definition, $\overline{p\ A\ e} \sim \bar{e}$, and so we know that $(p\ A\ e, e') \in \hat{R}$. This implies $p\ A\ e \Downarrow u$ such that $(u, v') \in R$. Therefore $(p\ A\ e, e') \in \hat{R}$ as desired. □

We can then use our lemma to show the following expected property:

Proposition 1. *(All Units Equal One Another) The following axiom is sound:*

$$\Pi x : 1, y : 1.\ x =_1 y$$

Proof. By extensionality and our lemma, we know that all $p : 1$ equal the polymorphic identity function, and then by transitivity the conclusion follows. □

In our proof sketches in the remainder of this section, we will elide contexts and environments, and furthermore equivocate between types and their relational interpretations, and between value and expression relations, in order to improve readability.

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

$$
\begin{array}{lcl}
\langle\rangle \sim_{\cdot}\ \langle\rangle & \iff & \text{always} \\
(\gamma_1, \overline{e_1}/x) \sim_{(\Gamma, x:X)} (\gamma_2, \overline{e_2}/x) & \iff & \gamma_1 \sim_\Gamma \gamma_2 \wedge \overline{e_1} \sim_{[\![\Gamma \vdash X:*]\!]\ \gamma_1} \overline{e_2} \\
(\gamma_1, (D_1, E_1, R_1)/x) \sim_{(\Gamma, \alpha:\kappa)} (\gamma_2, (D_2, E_2, R_2)/x) & \iff & \gamma_1 \sim_\Gamma \gamma_2 \wedge R_1 = R_2
\end{array}
$$

Figure 10. Environment Semantics

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

Figure 12. Kind Semantics

$$
\begin{array}{lcl}
[\![\Gamma, \alpha : \kappa, \Gamma' \vdash \alpha : \kappa]\!]\ (\gamma, (A, B, R)/\alpha, \gamma') & = & R \\
[\![\Gamma \vdash \lambda\alpha : \kappa.\ A : \Pi\alpha : \kappa.\ \kappa']\!]\ \gamma & = & \lambda(B_1, B_2, R).\ [\![\Gamma, \alpha : \kappa \vdash A : \kappa']\!]\ (\gamma, (B_1, B_2, R)/\alpha) \\
[\![\Gamma \vdash A\ B : [B/\alpha]\kappa']\!]\ \gamma & = & [\![\Gamma \vdash A : \Pi\alpha : \kappa.\ \kappa']\!]\ \gamma\ (\overline{\gamma(B)}, [\![\Gamma \vdash B : \kappa]\!]\ \gamma) \\
[\![\Gamma \vdash \lambda x : X.\ A : \Pi x : X.\ \kappa]\!]\ \gamma & = & \lambda\bar{e}.\ [\![\Gamma, x : X \vdash A : \kappa]\!]\ (\gamma, \bar{e}/x) \\
[\![\Gamma \vdash A\ e : [e/x]\kappa]\!]\ \gamma & = & [\![\Gamma \vdash A : \Pi x : X.\ \kappa]\!]\ \overline{\gamma(e)} \\
[\![\Gamma \vdash A : \kappa]\!]\ \gamma & = & [\![\Gamma \vdash A : \kappa']\!]\,(\text{when } \Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind})
\end{array}
$$

$$
\begin{array}{l}
[\![\Gamma \vdash \Pi x : X.\ Y : *]\!]\ \gamma = \\
\quad \left\{(\lambda x : X_1.\ e, \lambda x : X_1'.\ e') \;\middle|\; \begin{array}{l} \forall\,(e_0, e_0') \in \mathcal{E}\,[\![\Gamma \vdash X : *]\!]\ \gamma. \\ \quad ([e_0/x]e, [e_0'/x]e') \in \mathcal{E}\,[\![\Gamma, x : X \vdash Y : *]\!]\ (\gamma, (e_0, e_0')/x) \end{array}\right\} \\
[\![\Gamma \vdash \forall\alpha : \kappa.\ X : *]\!]\ \gamma = \\
\quad \left\{(\Lambda\alpha : \kappa_1.\ e, \Lambda\alpha : \kappa_1'.\ e') \;\middle|\; \begin{array}{l} \forall A, A', R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma. \\ \quad ([A/\alpha]e, [A'/\alpha]e') \in \mathcal{E}\,[\![\Gamma, \alpha : \kappa \vdash X : *]\!]\ (\gamma, (A, A', R)/\alpha) \end{array}\right\} \\
[\![\Gamma \vdash e_1 =_X e_2 : *]\!]\ \gamma = \\
\quad \{(\mathsf{refl}, \mathsf{refl}) \mid (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}\,[\![\Gamma \vdash X : *]\!]\ \gamma\}
\end{array}
$$

$$\mathcal{E}\,[\![\Gamma \vdash X : *]\!]\ \gamma = \left\{(e, e') \mid \exists v, v'.\ e \Downarrow v \wedge e' \Downarrow v' \wedge (v, v') \in [\![\Gamma \vdash X : *]\!]\ \gamma\right\}$$

Figure 13. Type Constructor Semantics

4.2 Existential Types

There is a standard encoding of existential types in terms of universals:

$$\exists\alpha : \kappa.\ Y = \forall\beta : *.\ (\forall\alpha : \kappa.\ Y \to \beta) \to \beta$$

The operations to form and use this type are

$$
\begin{array}{l}
\mathsf{pack} :\ \forall\alpha : \kappa.\ Y \to \exists\alpha : \kappa.\ Y \\
\mathsf{pack}\ \alpha\ x = \Lambda\beta.\ \lambda k : (\forall\alpha : \kappa.\ Y \to \beta).\ k\ \alpha\ x
\end{array}
$$

$$
\begin{array}{l}
\mathsf{unpack} :\ \exists\alpha : \kappa.\ Y \to \forall\beta : *.\ (\forall\alpha : \kappa.\ Y \to \beta) \to \beta \\
\mathsf{unpack}\ x\ \beta\ k = x\ \beta\ k
\end{array}
$$

The beta-rule for existentials $\mathsf{unpack}\ (\mathsf{pack}\ \alpha\ x)\ \beta\ k = k\ \alpha\ x$ is derivable without using parametricity. The commuting conversion and eta-rules, however, require parametricity to show sound.

Proposition 2. *(Extensionality for existentials)*

1. *For $t : \exists\alpha : \kappa.\ Y \to B$ and $e : \exists\alpha : \kappa.\ Y$, we have the commuting conversion $t\ (e\ (\exists\alpha : \kappa.\ Y)\ \mathsf{pack}) = e\ B\ (\Lambda\alpha : \kappa.\ \lambda y : Y.\ t\ (\mathsf{pack}\ \alpha\ y))$.*
2. *For $e : \exists\alpha : \kappa.\ Y$, we have $e\ (\exists\alpha : \kappa.\ Y)\ \mathsf{pack} = e$.*

Proof. We proceed as follows:

1. First define the functional relation
$$R = \left\{(v, b') \mid \bar{v} \in \exists\alpha : \kappa.\ Y \wedge \bar{b} \in B \wedge \overline{t\ v} \sim_B \bar{b}\right\}.$$
Then, by parametricity, we know that $(e\ (\exists\alpha : \kappa.\ Y), e\ B)$ are related at the type $(\forall\alpha : \kappa.\ Y \to R) \to R$. Now, we want to apply the terms $(\mathsf{pack}, \Lambda\alpha.\ \lambda y.\ t(\mathsf{pack}\ \alpha\ y))$, so we have to show they are in $\forall\alpha : \kappa.\ Y \to R$. Given an arbitrary $\overline{Z}$ and S, and a pair $\bar{y} \in [S/\alpha]Y$, we need to show that $(\mathsf{pack}\ Z\ y, t\ (\mathsf{pack}\ Z'\ y')) \in R$. From the definition of pack, this is equivalent to showing $(t\ (\mathsf{pack}\ Z\ y), t\ (\mathsf{pack}\ Z'\ y')) \in B$, which follows at once.

2. We want to show that $(e\ (\exists\alpha : \kappa.\ Y)\ \mathsf{pack}, e)$ are related. To show this, we assume we have a type $\overline{C}, R$ and continuations $\bar{k} \in \forall\alpha : \kappa.\ Y \to R$. Now, we define the relation

$$S \triangleq \left\{(r, q') \mid \bar{r} \in R \wedge \bar{q} \in \exists\alpha : \kappa.\ Y \wedge \bar{r} \sim_R \overline{q\ C\ k}\right\}$$

By parametricity, $(e\ (\exists\alpha : \kappa.\ Y), e\ C') \in (\forall\alpha : \kappa.\ Y \to S) \to S$. Now, we want to show that $(k, \mathsf{pack}) \in \forall\alpha : \kappa.\ Y \to S$. To show this, assume we have a $\overline{Z}, T$ and $\bar{y} \in [T/\alpha]Y$. So, we want to show that $(k\ Z\ y, \mathsf{pack}\ Z'\ y') \in S$. To show this, observe that $\overline{k\ Z\ y} \in R$ and $\overline{\mathsf{pack}\ Z\ y} \in \exists\alpha : \kappa.\ Y$. So it remains to be shown that $(k\ Z\ y, \mathsf{pack}\ Z'\ y'\ C'\ k') \in R$. But from the definition of pack, we get the goal $(k\ Z\ y, k'\ Z'\ y') \in R$, which we know. From this our goal follows.

□

4.3 Dependent Records

Cartesian products can be defined in the usual way, and there are no surprises with them. Significantly more interesting are dependent records. There is an obvious way of defining dependent records.

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

with an 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 (e.g., [13]) that once we have a suitable parametricity axiom, we can actually define *strong* eliminators for this type!

Proposition 3. *(Parametricity Axioms for Dependent Records) The following two axioms are sound:*

1. $\mathsf{ax}_1 : \Pi p : (\Sigma x : X.\ Y).\ p = p\ (\Sigma x : X.\ Y)\ \mathsf{pair}$
2. $\mathsf{ax}_2 : \Pi p : (\Sigma x : X.\ Y).\ \Sigma x : X.\ \Sigma y : Y.\ p = \mathsf{pair}\ x\ y$

Proof. Both of these properties rely on parametricity, and the second uses the first as a lemma.

1. Assume we have a pair $\bar{p} \in \Sigma x : X.\ Y$. Then, for all types $(\overline{B}, R)$ for α and continuations $\bar{k} \in \Pi x : X, y : Y.\ R$, we want to show $(p\ B\ k, p'\ (\Sigma x : X.\ Y)\ \mathsf{pair}\ B'\ k') \in R$. To do so, we define the following quasi-PER:

$$S \triangleq \left\{(r, q') \mid \bar{r} \in R \wedge \bar{q} \in \Sigma x : X.\ Y \wedge \overline{q\ B\ k} \sim_R \bar{r}\right\}$$

Hence $(p\ B, p\ (\Sigma x : X.\ Y)) \in (\Pi x : X, y : Y.\ S) \to S$. Then, we want to show that $(k, \mathsf{pair}) \in \Pi x : X, y : Y.\ S$. To show this, assume $\bar{x} \in X$ and $\bar{y} \in Y$. Now we must show that $(k\ x\ y, \mathsf{pair}\ x\ y) \in S$. By hypothesis, $\overline{k\ x\ y} \in R$ and by definition $\overline{\mathsf{pair}\ x\ y} \in \Sigma x : X.\ Y$. So it remains to show that $k\ x\ y \sim_R \mathsf{pair}\ x'\ y'\ B'\ k'$. But this follows by unfolding definitions. Hence $(p\ B\ k, p'\ (\Sigma x : X.\ Y)\ \mathsf{pair}) \in S$. Therefore it follows that $p\ B\ k \sim_R p'\ (\Sigma x : X.\ Y)\ \mathsf{pair}\ B'\ k'$, and so $p \sim_{\Sigma x:X.\ Y} p'\ (\Sigma x : X.\ Y)\ \mathsf{pair}$.

2. To show this, we will first show that for pairs $\bar{p} \in \Sigma x : X.\ Y$, the pair of terms

$$\overline{p\ (\Sigma x, y.\ p\ S\ \mathsf{pair} = \mathsf{pair}\ x\ y)\ (\lambda x\ y.\ \mathsf{pair}\ x\ (\mathsf{pair}\ y\ \mathsf{refl}))}$$

realizes the type $\Sigma x, y.\ p\ S\ \mathsf{pair} = \mathsf{pair}\ x\ y$. To save space, we have written S for the type $\Sigma x : X.\ Y$. We will also write $\hat{S}$ for its interpretation, and similarly we will write $T(z)$ for $\Sigma x : X.\ \Sigma y : Y.\ z\ S\ \mathsf{pair} = \mathsf{pair}\ x\ y$, and $\hat{T}(\bar{q})$ for its relational interpretation.
Now, note that since $\bar{p}$ is parametric, we can instantiate it at any relation we need. Now, we will choose the relation:

$$S' = \left\{\bar{q} \in \hat{S} \;\middle|\; \overline{q\ (T\ q)\ (\lambda x\ y.\ \mathsf{pair}\ x\ (\mathsf{pair}\ y\ \mathsf{refl}))} \in \hat{T}(\bar{q})\right\}$$

By unwinding definitions, we get that $\overline{\mathsf{pair}} \in \Pi x : X.\ Y \to S'$. Hence we can conclude that $\overline{p\ S\ \mathsf{pair}} \in S'$. Therefore, we know that $\overline{p\ S\ \mathsf{pair}\ T\ (\lambda x\ y.\ \mathsf{pair}\ x\ (\mathsf{pair}\ y\ \mathsf{refl}))} \in \hat{T}$. So this pair realizes the type $\Sigma x : X.\ \Sigma y : Y.\ p\ S\ \mathsf{pair} = \mathsf{pair}\ x\ y$.
Equality reflection using the previous lemma means that this axiom is also a proof for the type $\Sigma x : X.\ \Sigma y : Y.\ p = \mathsf{pair}\ x\ y$. As an aside, the fact that we can do this rewrite is a nice illustration of why having transitivity is useful in relational reasoning.

□

Once we have these axioms, it is possible to define the following operator:

$$\begin{array}{lcl}
 & & \forall C : (\Sigma x : X.\ Y) \to *. \\
\mathsf{repack} & : & \quad (\Pi x : X, y : Y.\ C(\mathsf{pair}\ x\ y)) \to \\
 & & \quad \Pi p : \Sigma x : X.\ Y.\ C(p) \\
\mathsf{repack}\ C\ f\ p & = & \mathsf{let}\ (x, q) = \mathsf{ax}_2\ p\ \mathsf{in} \\
 & & \mathsf{let}\ (y, pf) = q\ \mathsf{in} \\
 & & f\ x\ y
\end{array}$$

Here, we use the weak eliminators to make the proof term more readable. We use the ax_2 axiom to get a proof that the argument p is equal to $\mathsf{pair}\ x\ y$ for some x and y, and then unpack twice to get the x, the y, and the proof pf. Now, by hypothesis the type of $f\ x\ y$ is $C\ (\mathsf{pair}\ x\ y)$, and so by equality reflection on pf it also has the type $C(p)$.

Now we can use repack to define the projective eliminators:

$$\begin{array}{lcl}
\pi_1 & : & \Sigma x : X.\ Y \to X \\
\pi_1\ p & = & p\ X\ (\lambda x\ y.\ x) \\
 & & \\
\pi_2 & : & \Sigma x : X.\ Y \to [\pi_1(p)/x]Y \\
\pi_2\ p & = & \mathsf{repack}\ (\lambda q : \Sigma x : X.\ Y.\ [\pi_1(q)/x]Y) \\
 & & \quad (\lambda x\ y.\ y) \\
 & & \quad p
\end{array}$$

The definition of π_1 is straightforward, but the definition of the second projection is a little trickier. The second argument has the type $\Pi x : X, y : Y.\ \ Y \equiv \Pi x : X, y : Y.\ \ [\pi_1(\mathsf{pair}\ x\ y)/x]Y$, which can then be retyped using repack to the right type for the projective elimination.

4.4 Quotient Types

While not an application of parametricity in the sense of theorems for free [31], we can also show the realizability of quotient types [15] in our semantics. Quotient types, like their name suggests, are a way of defining new types by taking an existing type, and quotienting it by an equivalence relation.

To do this, we first define the auxilliary predicate Eq_X, which formalizes the notion of an equivalence relation. This is a predicate on relations of kind $X \to X \to *$, defined as follows:

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

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

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

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 $inj\ a = inj\ a'$. 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 $inj\ a = inj\ a'$. Second, if we lift a function f to operate on quotients, and we pass it the argument $inj\ x$, then the application of the lifted function should equal $f\ x$.

Proof. (Sketch) The proof of the soundness of the axiom is quite easy. Essentially, we just need to define the following relation:

$$S = \left\{ (v_1, v_2') \;\middle|\; \begin{array}{l} \exists v_1', v_2, \bar{q}. \\ \quad (v_1, v_1') \in X \wedge (v_2, v_2') \in X \wedge \\ \quad \bar{q} \in R\ (v_1, v_1')\ (v_2, v_2') \end{array} \right\}$$

Now, we can define the operators $\mathsf{inj} = \lambda x : X.\ x$ and $\mathsf{app} = \Lambda \gamma : *.\ \lambda f : X \to \gamma, pf : \ldots, x : X.\ f\ x$. Given these, we can then show the realizability of the term:

$\mathsf{pack}(X, \mathsf{pair\ inj}\ (\mathsf{pair\ app}\ (\mathsf{pair}\ (\lambda a, a', r.\ \mathsf{refl})\ (\Lambda \gamma.\ \lambda f, pf, x.\ \mathsf{refl}))))$

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. Just as an ML programmer might expect, we do not need to perform any changes of representation to protect the invariant of the quotient type — data abstraction is enough.

5. Soundness

Our main theorem is a consistency proof of our semantics. By an induction over derivations, we show that every well-typed expression lies in the expression relation. As a result, we know that the system is consistent: every closed term reduces to a value, and hence empty types are not inhabited.

Conceptually, we prove soundness in a series of stages, even though in our actual proof, all the lemmas are proven at once in a massive mutual induction. The full proof can be found online in the companion technical appendix at the authors' webpage [21].

5.1 Structural Lemmas

We begin the soundness theorem with a series of proofs that basic structural operations like weakening and strengthening lemmas of unused variables do not affect meaning in significant ways. For each of the context well-formedness, kind well-formedness and kinding judgements, weakening and strengthening hold.

The need for these properties arises from the interaction of the interpretation of type variables in the type semantics in Figure 13, and the interpretation of contexts given in Figure 10. When we look up a variable α in a substitution $\gamma \equiv (\gamma_0, (\overline{A}, R)/\alpha, \gamma_1) \in [\![\Gamma_0, \alpha : \kappa, \Gamma_1\ \mathsf{ok}]\!]$, the semantics of contexts means that $\gamma(\alpha) \in [\![\Gamma_0 \vdash \kappa : \mathsf{kind}]\!]\ \gamma_0$. Since Γ_0 is only a prefix of Γ, we need a theorem licensing us to treat $\gamma(\alpha)$ as an element of κ in the extended context $[\![\Gamma_0, \alpha : \kappa, \Gamma_1 \vdash \kappa : \mathsf{kind}]\!]\ \gamma$.

Lemma 2 (Context Weakening)**.** *Suppose that* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$. *Then*

1. *If* $R \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma$ *and* $\alpha \notin \mathrm{dom}(\Gamma, \Gamma')$, *then* $(\gamma, (\overline{D}, R)/\alpha, \gamma') \in [\![\Gamma, \alpha : \kappa, \Gamma'\ \mathsf{ok}]\!]$.
2. *If* $\bar{e} \in [\![\Gamma \vdash X : *]\!]\ \gamma$ *and* $x \notin \mathrm{dom}(\Gamma, \Gamma')$, *then* $(\gamma, \bar{e}/x, \gamma') \in [\![\Gamma, x : X, \Gamma'\ \mathsf{ok}]\!]$.

Lemma 3 (Context Strengthening)**.** *Suppose that* $\Gamma, \Gamma'\ \mathsf{ok}$. *Then:*

1. *If* $(\gamma, (\overline{D}, R)/\alpha, \gamma') \in [\![\Gamma, \alpha : \kappa, \Gamma'\ \mathsf{ok}]\!]$, *then* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$.
2. *If* $(\gamma, \bar{e}/x, \gamma') \in [\![\Gamma, x : X, \Gamma'\ \mathsf{ok}]\!]$, *then* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$.

Lemma 4 (Variable Irrelevance for Kinds)**.** *We have that:*

1. *If* $(\gamma, (\overline{D}, R)/\alpha, \gamma') \in [\![\Gamma, \alpha : \kappa, \Gamma'\ \mathsf{ok}]\!]$ *and* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$, *then* $[\![\Gamma, \alpha : \kappa', \Gamma \vdash \kappa' : \mathsf{kind}]\!]\ (\gamma, (\overline{D}, R)/\alpha, \gamma') = [\![\Gamma, \Gamma' \vdash \kappa' : \mathsf{kind}]\!]\ (\gamma, \gamma')$.
2. *If* $(\gamma, \bar{e}/x, \gamma') \in [\![\Gamma, x : X, \Gamma'\ \mathsf{ok}]\!]$ *and* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$, *then* $[\![\Gamma, x : X, \Gamma' \vdash \kappa' : \mathsf{kind}]\!]\ (\gamma, \bar{e}/x, \gamma') = [\![\Gamma, \Gamma' \vdash \kappa' : \mathsf{kind}]\!]\ (\gamma, \gamma')$.

Lemma 5 (Variable Irrelevance for Types)**.** *We have that:*

1. *If* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$ *and* $(\gamma, (\overline{D}, R)/\alpha, \gamma') \in [\![\Gamma, \alpha : \kappa, \Gamma'\ \mathsf{ok}]\!]$, *then* $[\![\Gamma, \Gamma' \vdash A : \kappa']\!]\ (\gamma, \gamma') = [\![\Gamma, \alpha : \kappa, \Gamma' \vdash A : \kappa']\!]\ (\gamma, (\overline{D}, R)/\alpha, \gamma')$.
2. *If* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$ *and* $(\gamma, \bar{e}/x, \gamma') \in [\![\Gamma, x : X, \Gamma'\ \mathsf{ok}]\!]$, *then* $[\![\Gamma, \Gamma' \vdash A : \kappa']\!]\ (\gamma, \gamma') = [\![\Gamma, x : X, \Gamma' \vdash A : \kappa']\!]\ (\gamma, \bar{e}/x, \gamma')$.

Lemma 6 (Variable Irrelevance for Expressions)**.** *We have that:*

1. *If* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$ *and* $(\gamma, (\overline{D}, R)/\alpha, \gamma') \in [\![\Gamma, \alpha : \kappa, \Gamma'\ \mathsf{ok}]\!]$, *then* $\mathcal{E}\ [\![\Gamma, \Gamma' \vdash X : *]\!]\ (\gamma, \gamma') = \mathcal{E}\ [\![\Gamma, \alpha : \kappa, \Gamma' \vdash X : *]\!]\ (\gamma, (\overline{D}, R)/\alpha, \gamma')$.
2. *If* $(\gamma, \gamma') \in [\![\Gamma, \Gamma'\ \mathsf{ok}]\!]$ *and* $(\gamma, \bar{e}/x, \gamma') \in [\![\Gamma, x : X, \Gamma'\ \mathsf{ok}]\!]$, *then* $\mathcal{E}\ [\![\Gamma, \Gamma' \vdash X : *]\!]\ (\gamma, \gamma') = \mathcal{E}\ [\![\Gamma, x : X, \Gamma' \vdash X : *]\!]\ (\gamma, \bar{e}/x, \gamma')$.

5.2 Stability

Next, we need to prove the stability properties.

To understand the intuition behind these properties, suppose that R is the relational interpretation of a type. Furthermore, suppose that we know that $(v_1, v_1') \in R$ and $(v_2, v_2') \in R$. Now, if $(v_1, v_1') \sim_R (v_2, v_2')$, then we know there are zig-zags connecting the two pairs of values. Therefore, they should be regarded as equivalent, and our semantics must not be able to tell the difference between them.

Another way of putting this is that any quasi-PER R can be divided into a disjoint collection S of non-empty, non-overlapping subsets of R, with the property that for each X in S, if $(x, y) \in X$ and $(x', y') \in X$, then $(x, y) \sim_R (x', y')$. So we can view a quasi-PER as a collection of "abstract values", and our semantics ought to not distinguish between different representatives of those abstract values.

The relation between substitutions, $\gamma \sim_\Gamma \gamma'$ essentially just lifts the quasi-PERs interpreting types over substitutions, so that we get a notion of equivalence for substitutions, as well. So our interpretations ought to be invariant under equivalent substitutions.

Lemma 7 (Stability of Kind Interpretation)**.** *If* $\Gamma\ \mathsf{ok}$ *and* $\Gamma \vdash \kappa : \mathsf{kind}$, *then for all* $\gamma, \gamma' \in [\![\Gamma\ \mathsf{ok}]\!]$ *such that* $\gamma \sim_\Gamma \gamma'$, $[\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma = [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma'$.

Lemma 8 (Stability of Type Interpretation)**.** *If* Γ ok *and* $\Gamma \vdash A : \kappa$*, then for all* $\gamma, \gamma' \in [\![\Gamma \text{ ok}]\!]$ *such that* $\gamma \sim_\Gamma \gamma'$*,* $[\![\Gamma \vdash A : \kappa]\!]\ \gamma = [\![\Gamma \vdash A : \kappa]\!]\ \gamma'$.

For expressions, we need to prove a somewhat stronger property. Since a substitution maps terms to variables, if we apply equivalent substitutions to the same term, the two substitutions ought to yield related pairs of terms.

Lemma 9 (Stability of Expressions)**.** *If* Γ ok *and* $\Gamma \vdash e : Y$*, then for all* $\gamma, \gamma' \in [\![\Gamma \text{ ok}]\!]$ *such that* $\gamma \sim_\Gamma \gamma'$*,* $\overline{\gamma(e)} \sim_{\mathcal{E}[\![\Gamma \vdash Y : *]\!]\ \gamma} \overline{\gamma'(e)}$.

An alternative approach to proving stability theorems would have been to define the semantics directly on the induced equivalence classes of terms. It was not clear to us whether that would save much work, so we chose to stick with more elementary methods.

5.3 Coherence

Because the typing rules for expressions and types have a conversion rule, we do not have a unique derivation property for well-formed terms. Furthermore, our semantic functions are defined on derivations, rather than on terms. As a result, we have to prove that the interpretations of kinds and types are coherent — regardless of how a well-formedness condition was derived, we have to ensure that the meaning comes out to be the same.

Lemma 10 (Coherence of Kind Interpretation)**.** *If* Γ ok *and* $\Gamma \vdash \kappa : \mathsf{kind}$ *and* $\Gamma \vdash \kappa' : \mathsf{kind}$*, then* $[\![\Gamma \vdash \kappa : \mathsf{kind}]\!] = [\![\Gamma \vdash \kappa' : \mathsf{kind}]\!]$.

Lemma 11 (Coherence of Type Interpretation)**.** *If* Γ ok *and* $\Gamma \vdash A : \kappa$ *and* $\Gamma \vdash A : \kappa'$*, then* $[\![\Gamma \vdash A : \kappa]\!] = [\![\Gamma \vdash A : \kappa']\!]$.

Since the interpretation of types of kind $*$ is a value relation, we have to show that value relations lifted to expression relations are also coherent.

Lemma 12 (Coherence of Expression Interpretation)**.** *If* Γ ok *and we have two derivations* $D :: \Gamma \vdash X : *$ *and* $D' :: \Gamma \vdash X : *$*, then* $\mathcal{E}\,[\![D :: \Gamma \vdash X : *]\!] = \mathcal{E}\,[\![D' :: \Gamma \vdash X : *]\!]$.

5.4 Substitution

Next, we have to prove the soundness of substitution, which simply means that the semantics and the syntax agree about the meanings of substituted terms. We begin by showing that substitution lets us move types and terms into and out of an environment. Since substitutions map closed terms for expressions, we do not need to perform a "substitution on the substitution" when we add or remove bindings.

Theorem 1 (Soundness of Substitution Into Contexts)**.** *We have that:*

1. *If* $\Gamma \vdash A : \kappa$ *then* $(\gamma, (\overline{\gamma(P)}, [\![\Gamma \vdash A : \kappa]\!]\ \gamma)/\alpha, \gamma') \in [\![\Gamma, \alpha : \kappa, \Gamma' \text{ ok}]\!]$ *if and only if* $(\gamma, \gamma') \in [\![\Gamma, [A/\alpha]\Gamma' \text{ ok}]\!]$.
2. *If* $\Gamma \vdash e : X$ *then* $(\gamma, \overline{\gamma(e)}/x, \gamma') \in [\![\Gamma, x : X, \Gamma' \text{ ok}]\!]$ *if and only if* $(\gamma, \gamma') \in [\![\Gamma, [e/x]\Gamma' \text{ ok}]\!]$.

After that, we can show that we can substitute well-kinded types and well-typed expressions into kinds, types and expressions.

Theorem 2 (Soundness of Substitution Into Kinds)**.** *We have that:*

1. *If* $\Gamma \vdash A : \kappa$ *then* $[\![\Gamma, \alpha : \kappa, \Gamma' \vdash \kappa_1 : \mathsf{kind}]\!]\ (\gamma, [\![\Gamma \vdash A : \kappa]\!]\ \gamma/\alpha, \gamma') = [\![\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]\kappa_1 : \mathsf{kind}]\!]\ (\gamma, \gamma')$.
2. *If* $\Gamma \vdash e : X$ *then* $[\![\Gamma, x : X, \Gamma' \vdash \kappa_1 : \mathsf{kind}]\!]\ (\gamma, \overline{\gamma(e)}/x, \gamma') = [\![\Gamma, [e/x]\Gamma' \vdash [e/x]\kappa_1 : \mathsf{kind}]\!]\ (\gamma, \gamma')$.

Theorem 3 (Soundness of Substitution Into Types)**.** *We have that:*

1. *If* $\Gamma \vdash A : \kappa$ *then* $[\![\Gamma, \alpha : \kappa, \Gamma' \vdash B : \kappa_1]\!]\ (\gamma, (\overline{\gamma(A)}, [\![\Gamma \vdash A : \kappa]\!]\ \gamma)/\alpha, \gamma') = [\![\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]B : \kappa_1]\!]\ (\gamma, \gamma')$.
2. *If* $\Gamma \vdash e : X$ *then* $[\![\Gamma, x : X, \Gamma' \vdash B : \kappa_1]\!]\ (\gamma, \overline{\gamma(e)}/x, \gamma') = [\![\Gamma, [e/x]\Gamma' \vdash [e/x]B : \kappa_1]\!]\ (\gamma, \gamma')$.

Theorem 4 (Soundness of Substitution Into the Expression Relation)**.** *We have that:*

1. *If* $\Gamma \vdash A : \kappa$ *then* $\mathcal{E}\,[\![\Gamma, \alpha : \kappa, \Gamma' \vdash Y : *]\!]\ (\gamma, (\overline{\gamma(A)}, [\![\Gamma \vdash A : \kappa]\!]\ \gamma)/\alpha, \gamma') = \mathcal{E}\,[\![\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]Y : *]\!]\ (\gamma, \gamma')$.
2. *If* $\Gamma \vdash e : X$ *then* $\mathcal{E}\,[\![\Gamma, x : X, \Gamma' \vdash Y : *]\!]\ (\gamma, \overline{\gamma(e)}/x, \gamma') = \mathcal{E}\,[\![\Gamma, [e/x]\Gamma' \vdash [e/x]Y : *]\!]\ (\gamma, \gamma')$.

We do not need to prove a similar substitution lemma for expressions, since all the properties we need follow from stability and the basic properties of substitutions in the lambda-calculus.

5.5 Semantic Soundness

Finally, we can state the soundness theorems for our language. We begin with the well-definedness of our basic judgements:

Theorem 5 (Soundness of Context Interpretation)**.** *If* Γ ok*, then* $[\![\Gamma \text{ ok}]\!] \in \mathrm{Set}$.

Theorem 6 (Soundness of Kind Interpretation)**.** *If* Γ ok *and* $\Gamma \vdash \kappa : \mathsf{kind}$*, then* $[\![\Gamma \vdash \kappa : \mathsf{kind}]\!] \in [\![\Gamma \text{ ok}]\!] \to \mathrm{Set}$.

Theorem 7 (Soundness of Type Interpretation)**.** *If* Γ ok *and* $\Gamma \vdash A : \kappa$*, then* $[\![\Gamma \vdash A : \kappa]\!] \in \Pi\gamma \in [\![\Gamma \text{ ok}]\!].\ [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma$.

Theorem 8 (Soundness of Expression Interpretation)**.** *If* Γ ok *and* $\Gamma \vdash Y : *$*, then for all* $\gamma \in [\![\Gamma \text{ ok}]\!]$*,* $\mathcal{E}\,[\![\Gamma \vdash Y : *]\!]\ \gamma \in \textsc{Qper}(\mathrm{Exp}, \mathrm{Exp})$.

Note that we could not even show that the interpretation of contexts or kinds was well-defined until we had the rest of the metatheory in place, due to the circularities induced by type dependency.

Next, we show the fundamental property of logical relations, which says that for every well-formed substitution γ and well-typed term e, the pair of terms $\overline{\gamma(e)}$ is in the expression relation at that type.

Theorem 9 (Fundamental Property)**.** *If* Γ ok *and* $\Gamma \vdash e : X$*, then for all* $\gamma \in [\![\Gamma \text{ ok}]\!]$*,* $\overline{\gamma(e)} \in \mathcal{E}\,[\![\Gamma \vdash X : *]\!]\ \gamma$.

Next, we can show that the judgemental equality of the language is sound. For kinds and types, equality is set-theoretic equality.

Theorem 10 (Soundness of Kind Equality)**.** *If* Γ ok *and* $\Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}$*, then for all* $\gamma \in [\![\Gamma \text{ ok}]\!]$*,* $[\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma = [\![\Gamma \vdash \kappa' : \mathsf{kind}]\!]\ \gamma$.

Theorem 11 (Soundness of Type Equality)**.** *If* Γ ok *and* $\Gamma \vdash A \equiv B : \kappa$*, then for all* $\gamma \in [\![\Gamma \text{ ok}]\!]$*,* $[\![\Gamma \vdash A : \kappa]\!]\ \gamma = [\![\Gamma \vdash B : \kappa]\!]\ \gamma$.

On the other hand, equality for terms means that the substitution instances are related by the logical relation at that type.

Theorem 12 (Soundness of Term Equality)**.** *If* Γ ok *and* $\Gamma \vdash e_1 \equiv e_2 : X$*, then for all* $\gamma \in [\![\Gamma \text{ ok}]\!]$*,* $(\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}\,[\![\Gamma \vdash X : *]\!]\ \gamma$.

5.6 Consistency

An easy corollary of the soundness proof is that the language is terminating.

Corollary 1. *(Termination) If* $\cdot \vdash e : X$*, then* $e \Downarrow v$.

This entails that the langage is consistent – that is, we can conclude that there are no terms of type $\forall\alpha : *.\ \alpha$, since we could use it to type looping combinators.

6. Related and Future Work

Quasi-PERs The earliest use of quasi-PERs to to model data abstraction we have found is by Tennent and Takeyama [29], 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 "zig-zag complete relations").

The term "quasi-PER" was coined by Hofmann [16], 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 faced a representation independence problem in showing the equivalence of programs with possibly-differing sets of effects (i.e., what is called "effect masking" [22] in the literature on effects).

Our work vastly 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 wild array of type-theoretic features.

Hutton and Voermans [18] studied a relational version of Squiggol [9], in which the category of sets and relations was replaced with the category of PERs and saturated relations[1] between them. 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.

Semantic Models for Parametricity The standard approach to building parametric models (typically of F), is to begin with a non-parametric model of the language, and then to give an intepretation of types as certain relations over the non-parametric semantic types. This approach is used in Bainbridge et al. [5], and an abstract characterization of it was given by Robinson and Rosolini [27] using categories of *reflexive graphs*, which both Dunphy and Reddy [14] and Birkedal et al. [10] have developed further.

Our model does not immediately fit into this framework, since we directly give a relational semantics, *without* first building a non-parametric model. This represents a considerable simplification of the metatheory – we only need to build a semantics once, rather than twice. (This advantage is particularly acute given the complexity of semantics for dependent type theory.)

Our work builds a binary relational model, which is already enough to start proving rather interesting representation independence properties. However, it is well known that as the arity of the relations increase, the strength of parametricity properties increases.

The fully general version, dubbed "Kripke logical relations of varying arity", was introduced by Jung and Tiuryn [19] to characterize lambda-definability, and more recently Atkey [3] has used a similar Kripke model to show the correctness of parametric higher-order abstract syntax [11]. In the future, we hope to study whether it is possible to give direct versions of these models, as well.

Both Vytiniotis and Weirich [30] and Atkey [4] have each given parametric models of F_ω, which (as is well known) is equivalent to the calculus of constructions minus dependency. It is worth comparing these two approaches, since the differences between them shed some light on our own model. Vytiniotis and Weirich give a term model, much as we do, and as such they need to prove a coherence theorem for the interpretation of kinds. In contrast, Atkey 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 the underlying semantics with the relational semantics he defines.

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). Vytiniotis and Weirich take great care to ensure that the relations they define only speak about well-formed type constructors. We do not bother maintaining this invariant, since it is unnecessary: 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. (Intuitively, this is like working with untyped realizers.)

To even state the identity extension property for higher kinds, it is necessary to equip each kind with a distinguished notion of identity relation connecting the base and relational interpretations. We do not need to do this, since we have a purely relational semantics. However, we do need to ensure that our interpretation of higher kinds respects the equalities (quasi-PER) on types — this is what we dubbed the stability condition. We found this an easier condition to work with than defining identity relations at all kinds, since we only need to consider the effect of the base kind.

Internalizations of Parametricity In recent work, Bernardy et al. [8] have shown that it is possible to generalize Reynolds' relational semantics to systems of dependent types, and 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. Their work is complementary to our own.

The translational approach is wholly syntactic, as opposed to our more semantic approch. One benefit of their method is that it yields concrete proof terms for parametricity properties. However, two downsides of the translational approach are that (1) parametricity properties only apply to closed expressions, and cannot be applied to open ones, and (2) there is no way to use parametricity to internalize program equivalences.

In contrast, our approach does not give full proof terms, due to our use of equality reflection. However, we can make use of parametricity arguments even for open terms, and *can* internalize parametric program equivalences as equalities (as our examples with existentials and quotient types demonstrate).

As an alternative to our semantic approach to internalization, Bernardy and Moulin [7] have also investigated extending the syntax of type theory with operators to represent appeals to parametricity. Though they end up performing very radical surgery to the syntax of type theory (involving adding hypercubes of binders and types to describe relations), the fundamental idea is quite intuitive: they add an operator to type theory which asserts that all variables are parametric, and add enough other machinery to support shuffling proof terms around to ensure that proofs of this fact are available whenever they are needed.

In contrast, the fact that all variables are parametric is an intrinsic consequence of the fact that our model is parametric. As a result, we do not need to pass around this information. Furthermore, by restricting our relations to quasi-PERs, we guarantee that equivalence is transitive, which permits equating the identity type with parametricity. As a result, the statement of parametricity in our type theory is just the reflexivity of equality. Bernardy and Moulin's approach does not validate this equation: they allow inductive definitions in the style of Coq and Agda, which allows internalizing the conversion relation by defining the Martin-Löf identity type as an inductive type.

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 that we can support. There are grounds for hope: all

[1] A relation R between PERs A and B is *saturated* when it respects the PER structure on A and B. That is, if $A(a', a)$ and $R(a, b)$ and $B(b, b')$ then $R(a', b')$.

of the axioms we studied have *computational realizers*, even if they are not typable in the plain calculus of constructions.

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):428 – 469, 2006. ISSN 1570-8683.

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

[3] R. Atkey. Syntax for free: Representing syntax with binding using parametricity. In P.-L. Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.

[4] R. Atkey. Relational parametricity for higher kinds. In *Computer Science Logic 2012*, 2012. To Appear.

[5] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theor. Comput. Sci.*, 70(1):35–64, 1990.

[6] H. Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991.

[7] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *LICS*, pages 1–10. IEEE Computer Society, 2012.

[8] J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *ICFP 2010*, ICFP '10, pages 345–356, New York, NY, USA, 2010. ACM.

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

[10] L. Birkedal, R. E. Møgelberg, and R. L. Petersen. Domain-theoretical models of parametric polymorphism. *Theor. Comput. Sci.*, 388(1-3): 152–172, 2007.

[11] A. 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.

[12] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, Feb. 1988. ISSN 0890-5401.

[13] 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.

[14] B. P. Dunphy and U. S. Reddy. Parametric limits. In *LICS*, pages 242–251. IEEE Computer Society, 2004.

[15] M. Hofmann. A simple model for quotient types. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 216–234. Springer Berlin / Heidelberg, 1995. 10.1007/BFb0014055.

[16] 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.

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

[18] G. Hutton and E. Voermans. Making Functionality More General. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, Springer-Verlag Series of Workshops in Computing, Skye, Scotland, 1992.

[19] A. Jung and J. Tiuryn. A new characterization of lambda definability. *Typed Lambda Calculi and Applications*, pages 245–257, 1993.

[20] N. R. Krishnaswami and N. Benton. Adding equations to System F types. In *21st European Symposium on Programming (ESOP 2012)*, Lecture Notes in Computer Science. Springer-Verlag, March 2012.

[21] N. R. Krishnaswami and D. Dreyer. A relationally-parametric model of the calculus of constructions. Auxilliary materials at `http://www.mpi-sws.org/~neelk/paradep-techreport.pdf`, 2012.

[22] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7.

[23] P. Martin-Lof. *Intuitionistic type theory*. Bibliopolis Naples, Italy, 1984.

[24] J. McKinna. Why dependent types matter. In *POPL*, page 1, 2006.

[25] G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA*, pages 361–375, 1993.

[26] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[27] E. P. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *LICS*, pages 364–371. IEEE Computer Society, 1994.

[28] R. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95:33 – 48, 1984.

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

[30] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Functional Programming*, 20(2):175–210, March 2010. ISSN 0956-7968.

[31] P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0.