Relational Parametricity for Higher Kinds

Robert Atkey[1]

1 **University of Strathclyde, UK**
Robert.Atkey@strath.ac.uk

Abstract

Reynolds' notion of relational parametricity has been extremely influential and well studied for polymorphic programming languages and type theories based on System F. The extension of relational parametricity to higher kinded polymorphism, which allows quantification over type operators as well as types, has not received as much attention. We present a model of relational parametricity for System Fω, within the impredicative Calculus of Inductive Constructions, and show how it forms an instance of a general class of models defined by Hasegawa. We investigate some of the consequences of our model and show that it supports the definition of inductive types, indexed by an arbitrary kind, and with reasoning principles provided by initiality.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.3.2 Semantics of Programming Languages

Keywords and phrases Relational Parametricity, Higher Kinds, Polymorphism

1 Introduction

Reynolds defined relational parametricity to formalise the intuition that, in the absence of introspection capabilities, a polymorphic program must act uniformly in the choice of type instantiation [14]. The inability of a program to rely on details of data representation that it has not been explicitly exposed to forms the backbone of reasoning based on information hiding and abstraction. The core of Reynolds' idea is that a parametrically polymorphic program should preserve all relations between any pair of types that it is instantiated with.

Since Reynolds' definition, many interesting consequences have been revealed. Wadler called some of these consequences "Theorems for free" [20] where theorems can be stated about polymorphically typed programs just by looking at their types. It is also possible to prove that the polymorphic λ-calculus allows encodings of categorical constructions such as finite products and coproducts, initial algebras and final coalgebras, as long as the model is parametric in Reynolds' sense, as demonstrated by Hasegawa [7].

Most of the previous work on parametricity has been based on languages and type theories building on System F, where type quantification is only over types. Modern programming languages now include *higher kinded* polymorphism where programs may be parameterised by type operators of kinds like $* \to *$ as well as normal types of kind $*$. Type operators are simply functions operating on types, where the kinds act as a type system at "one level up" to classify types. The purely functional language Haskell uses type operators of kind $* \to *$ to represent constructions such as monads [11], and the GHC compiler has recently added support for user defined kinds [21]. The JVM-based language Scala [12] also includes support for type operators. Languages in the ML family such as OCaml and SML do not support higher kinded polymorphism in their core languages, but do support a form of it through their module systems. Indeed, Rossberg *et al.* [16] have shown that ML-style modules can be understood by translation into a language with higher kinds.

In this paper, we consider the prototypical calculus with type operators and higher kinded polymorphism, System Fω. We give a concrete model of this calculus within the impredicative

variant of the Calculus of Inductive Constructions, building on the interpretation of kinds as reflexive graphs, due to Hasegawa [8, 7] and Robinson and Rosolini [15]. The use of reflexive graphs builds Reynolds' identity extension principle into the interpretation, ensuring that we are able to perform hypothetical reasoning with parametric functions. We make use of this ability to show that we can represent κ indexed inductive types, for arbitrary kinds κ, within System Fω, and prove their initiality. Proving initiality is essential for reasoning about members of inductive types. When $\kappa = *$, we obtain a method for encoding and reasoning about Generalised Algebraic Data Types (GADTs) [5].

1.1 Background

To our knowledge, the earliest formulation of a theory of relational parametricity for higher kinded types is due to Hasegawa [8], building on his previous work on relationally parametric System F [7]. Hasegawa gives a category theoretic approach, based on interpreting kinds and types in categories of *r-frames*: sets equipped with a notion of reflexive relations between their elements, building in Reynolds' identity extension property. We recall Hasegawa's definition of r-frame in Section 3.3, and describe how it relates to our model construction, and to the reflexive graph approach of Robinson and Rosolini [15]. Hasegawa presents a set of general requirements on a model in order for it to interpret System Fω, and demonstrates two instances: a PER-based model and a syntactic model. Hasegawa also develops a number of applications of models, using notions of enriched category theory, including the existence of initial algebras and final coalgebras for a certain class of internally defined functors.

Takeuti [17] formulated a definition of relational parametricity for dependent type theories in the λ-cube, including System Fω, and developed a small amount of enriched category theory in this setting.

Bernardy, Jansson and Paterson [3] present a slightly different formulation of parametricity for pure type systems (PTSs), including those in the λ-cube. A key feature of this work is that the relational interpretation of a type is expressed within an augmented version of the PTS that is started with. Bernardy *et al.* prove an abstraction theorem for their translation, with the neat property that the same translation that translates types to relational interpretations also translates terms to the proof of the abstraction theorem for that term. This work does not take into account the identity extension aspect of parametricity, nor do the authors construct a model with the property that all members of universal types are relationally parametric, so the constructions we present in Section 4 are not possible in their setting.

Voigtländer [18] has examined an extension of free theorems to a fragment of System Fω, where he only considers functions with types of form $\forall_\kappa \alpha.\ A[\alpha] \to B[\alpha]$, where κ may be a higher kind. Voigtländer derives several useful free theorems relating functions that operate polymorphically over monads. We expect that all his free theorems are true in our model.

Vytiniotis and Weirich [19] present a syntactic model of parametricity for System Fω extended with representation types. They define a relational interpretation of types where great care is required, in their syntactic setting, to ensure that the interpretation is coherent under type equality. This is handled for us in our extensional denotational setting. Vytiniotis and Weirich consider an extended system with type representations and prove that, due to parametricity, run-time type casting using type representations will always be equivalent to the identity function. In comparison to the present work, they do not need to consider a definition of the interpretation of kinds that forces there to be a distinguished identity relation for every type, because they need only consider the types that are syntactically definable. We have also considered constructions within parametric System Fω beyond equality types.

$$\frac{\alpha : \kappa \in \Theta}{\Theta \vdash \alpha : \kappa} \qquad \frac{\Theta, \alpha : \kappa_1 \vdash A : \kappa_2}{\Theta \vdash \lambda\alpha : \kappa_1.\ A : \kappa_1 \to \kappa_2} \qquad \frac{\Theta \vdash F : \kappa_1 \to \kappa_2 \qquad \Theta \vdash A : \kappa_1}{\Theta \vdash FA : \kappa_2}$$

$$\frac{\Theta \vdash A : * \qquad \Theta \vdash B : *}{\Theta \vdash A \to B : *} \qquad \frac{\Theta, \alpha : \kappa \vdash A : *}{\Theta \vdash \forall_\kappa \alpha.\ A : *}$$

Figure 1 Types and their Kinds

1.2 Contributions of this Paper

- We define a concrete type theoretic model of relationally parametric System Fω, a polymorphic λ-calculus with higher kinded types. We give our model in direct terms, not using category theoretic language, in an attempt to make the definition more accessible. We also relate our model to a standard non-parametric semantics (Theorem 4), in order to show that the relationally parametric model can be used to reason about interpretations in the non-parametric model.
- We demonstrate that our relationally parametric model of System Fω allows for the definition of indexed inductive types, with an initiality property that allows for reasoning. Hasegawa also proves the existence of initial algebras in relationally parametric models of System Fω, for functors satisfying a condition he calls the *middle-lax* property. We demonstrate initial algebras for all definable functors, and present our proof in more elementary terms. We also demonstrate how to interpret data kinds, such as type-level natural numbers.

We emphasise that even though our model is constructed on paper with the impredicative Calculus of Inductive Constructions, we have not yet completed a full verification of all our results in Coq. Also, there are evidently many more constructions that one might consider in a relationally parametric model of System Fω, including final coalgebras. We have some preliminary results in this direction, but leave development of them to future work.

2 Type Polymorphism with Higher Kinds

To fix notation we define the syntax and typing of System Fω in this section, following the standard presentations [13]. There are three levels: *kinds*, which classify *types*, which classify *terms*. The language of kinds consists of simple types over a single base kind $*$ that represents all types that can classify terms. We use meta-syntactic variables $\kappa, \kappa_1, \kappa_2, ...$ to stand for kinds. The grammar $\kappa ::= * \mid \kappa_1 \to \kappa_2$ defines the collection of kinds that we consider. We will look at extending the calculus with additional kinds in Section 5.

The language of types is an applied simply-typed λ-calculus with constants for function and universal types. The kinding rules that classify types by kind are shown in Section 1. We will use the general name "type" for both proper types of kind $*$ and for higher kinded type operators. We use Θ as a meta-variable to stand for kinding contexts $\alpha_1 : \kappa_1, ..., \alpha_n : \kappa_n$, Greek letters $\alpha, \beta, ...$ for type variables in the calculus and capital roman letters $A, B, F, ...$ for meta-syntactic variables standing for arbitrary types. Types come equipped with an equational theory ($\equiv$) built from the standard typed $\beta\eta$ rules for the simply typed λ-calculus.

The typing rules for terms are shown in Section 2. These are the standard rules for System Fω. We assume throughout that all the types in the typing context Γ are of base

$$\frac{x : A \in \Gamma}{\Theta \mid \Gamma \vdash x : A} \qquad \frac{\Theta \mid \Gamma \vdash e : A \qquad \Theta \vdash A \equiv B : *}{\Theta \mid \Gamma \vdash e : B}$$

$$\frac{\Theta \mid \Gamma, x : A \vdash e : B}{\Theta \mid \Gamma \vdash \lambda x : A.\ e : A \to B} \qquad \frac{\Theta \mid \Gamma \vdash e_1 : A \to B \qquad \Theta \mid \Gamma \vdash e_2 : A}{\Theta \mid \Gamma \vdash e_1 e_2 : B}$$

$$\frac{\Theta, \alpha : \kappa \mid \Gamma \vdash e : A \qquad \alpha \notin fv(\Gamma)}{\Theta \mid \Gamma \vdash \Lambda\alpha : \kappa.\ e : \forall_\kappa \alpha.A} \qquad \frac{\Theta \mid \Gamma \vdash e : \forall_\kappa \alpha.A \qquad \Theta \vdash B : \kappa}{\Theta \mid \Gamma \vdash e\ [B] : A\{B/\alpha\}}$$

Figure 2 Terms and their Types

kind according to the kinding context Θ. We use the notation $A\{B/\alpha\}$ to represent capture avoiding substitution of B for all occurrences of α in A. The final typing rule imports the equational theory of types into the typing judgement, allowing a term to have many syntactically different types. Terms come equipped with an equational theory of their own: $\beta\eta$ equalities for λ-abstraction and Λ-abstraction.

3 A Relationally Parametric Model

We now present the construction of a relationally parametric model of System Fω in the impredicative Calculus of Inductive Constructions (CIC). CIC is a dependently typed λ-calculus with impredicative polymorphism and inductive types, and forms the basis of the Coq proof assistant. The presence of impredicative polymorphism in our meta-theory simplifies the presentation of our model, allows us to concentrate on the parametricity aspects, and permits straightforward reasoning inside the model. This technique has been used to construct models of System F with Kripke parametricity for the purposes of studying the adequacy of Higher Order Abstract Syntax (HOAS) encodings [1]. Our model is an instance of the class of models defined by Hasegawa, and also those defined by Robinson and Rosolini. We cover the connection in Section 3.3. We motivate this class of models in Section 3.2.

3.1 Setting up the Metatheory

Impredicative CIC allows us to quantify over all objects of sort Set to generate a new object of sort Set. This feature can be enabled in the Coq proof assistant by means of the `-impredicative-set` command line option. CIC already includes an impredicative sort of propositions Prop. Importantly, this is disjoint from the sort Set, where we build our denotations of types.

We require two additional axioms to be added to CIC. The first of these is extensionality for functions, which states that two functions are equal if they are equal for all inputs: $\forall A \in \mathsf{Type}, B \in A \to \mathsf{Type}, f, g \in (\forall a.Ba).\ (\forall x.\ fx = gx) \to f = g$. Extensionality for functions allows our denotational model to smoothly support the η equality rules of System Fω, without requiring complex constructions involving setoids. We also require propositional extensionality, which will allow us to treat equivalent propositions as equal: $\forall P, Q \in \mathsf{Prop},\ (P \leftrightarrow Q) \to P = Q$. Propositional extensionality implies proof irrelevance: all proofs of a given proposition are equal: $\forall P \in \mathsf{Prop}.\ \forall p_1, p_2 \in P.\ p_1 = p_2$. These axioms allow us to define data with embedded proofs that are equal when their computational parts are equal, which allows us to prove more equalities between denotations of types.

We informally justify our use of these axioms, plus impredicativity, by the existence of models of CIC in intuitionistic set theory. In the remainder of the paper, we use informal set theoretic notation and do not explicitly highlight the uses of these axioms. Note that everywhere we refer to "set"s, we mean CIC objects of sort Set.

3.2 Types, Relations and Identity Extension

We consider the extension of Reynolds' definition of relational parametricity to a type system with higher kinds, in order to motivate the model construction in the rest of this section.

If we model types of kind $*$ as sets, then the modelling of the rest of the kind hierarchy has an obvious choice: we interpret kinds of the form $\kappa_1 \to \kappa_2$ as functions:

$$[\![*]\!] = \mathsf{Set} \qquad\qquad [\![\kappa_1 \to \kappa_2]\!] = [\![\kappa_1]\!] \to [\![\kappa_2]\!]$$

To define relational parametricity, we need to consider what relations between the interpretations of types look like. At the base kind, $*$, this will be just the collection of all relations between a pair of sets: we denote the collection of relations between sets A and B as $\mathsf{Rel}(A, B)$. At higher kinds, an obvious approach is to consider relation transformers:

$$\begin{array}{rcl} [\![\kappa]\!]^{\mathcal{R}} & \in & [\![\kappa]\!] \times [\![\kappa]\!] \to \mathsf{Set} \\ [\![*]\!]^{\mathcal{R}} & = & (A, B) \mapsto \mathsf{Rel}(A, B) \\ [\![\kappa_1 \to \kappa_2]\!]^{\mathcal{R}} & = & (F, G) \mapsto \forall A, B \in [\![\kappa_1]\!].[\![\kappa_1]\!]^{\mathcal{R}}(A, B) \to [\![\kappa_2]\!]^{\mathcal{R}}(FA, GB) \end{array}$$

Following relationally parametric models of System F (e.g., Jacobs [9], Section 8.4), for any well-kinded type $\alpha_1 : \kappa_1, \cdots, \alpha_n : \kappa_n \vdash A : \kappa$, we will have an underlying interpretation $[\![A]\!]^f \in [\![\kappa_1]\!] \times \cdots \times [\![\kappa_n]\!] \to [\![\kappa]\!]$ and a relational interpretation $[\![A]\!]^r \in \forall \overrightarrow{A}, \overrightarrow{B}.[\![\kappa_1]\!]^R(A_1, B_1) \times \cdots \times [\![\kappa_n]\!]^R(A_n, B_n) \to [\![\kappa]\!]^R(F\overrightarrow{A}, F\overrightarrow{B})$.

However, this interpretation of kinds suffers from a problem. In order to allow for hypothetical reasoning using relational parametricity, we must ensure that the interpretation of a type of the form $\forall_\kappa \alpha.A$ only contains elements that actually preserve all relations (i.e., are parametric). If the type A is open—it has free type variables—then in order to state this property we need a default relational interpretation for each of the free type variables. In Reynolds' formulation of relational parametricity, where all type variables have kind $*$, the default relational interpretation is given by equality. Moreover, given a type with free type variables, substituting the equality relation for all the free variables in the relational interpretation of the type should give the equality relation: this is Reynolds' *identity extension* property, which is needed to prove the abstraction theorem (see Theorem 3, below).

In the situation with higher kinds, some of the variables may be of functional kinds like $\kappa_1 \to \kappa_2$, so to even state identity extension we need a notion of identity relation for any type at these kinds. We cannot use equality, because we require a relation transformer. Moreover, this transformer should obey the identity extension property itself; sending the identity relations for kind κ_1 to the identity relations for kind κ_2. But what are these identity relations? There does not seem to be an obvious way to assign an identity relation transformer to an arbitrary type operator, so we need to equip each element of $[\![\kappa_1 \to \kappa_2]\!]$ with a distinguished relation transformer with the identity extension property.

3.3 Kinds as Reflexive Graphs

Hasegawa [7, 8] and Robinson and Rosolini [15] give us the solution. Instead of defining the underlying and relational semantics of kinds separately, we treat each kind as a *reflexive*

graph. A reflexive graph consists of two collections: one of *objects* and one of *relations*. Every relation is assigned a source and target object, and every object a has a distinguished "identity" relation, whose source and target are a. Hasegawa calls such structures *r-frames*, and notes that they are essentially categories without composition. Robinson and Rosolini note that the category of reflexive graphs is just the category of presheaves over the category $\bullet \underset{\delta_1}{\overset{\delta_0}{\underset{\longleftarrow}{\overset{\longrightarrow}{\longleftarrow}}}} \bullet$ such that $\delta_0 \circ i = id$ and $\delta_1 \circ i = id$. It immediately follows that there is an interpretation of higher kinds, using the cartesian closed structure of this presheaf category, although Robinson and Rosolini do not make use of this fact. This solves the problem we identified above: the assignment of a distinguished identity relation to every inhabitant of the denotation of a higher kind. Dunphy and Reddy [6] also use reflexive graphs to study relational parametricity.

It the rest of this section, we present a concrete model of System Fω within CIC that interprets kinds as reflexive graphs. We do not make explicit use of the presheaf structure underlying the model, in order to keep the presentation straightforward and accessible.

3.4 Interpretation of Kinds

A kind κ is interpreted as a triple of a carrier A; a function that takes a pair of elements of A and returns the set of "relations" between them; and a special distinguished "identity" relation for every element of A. We write this specification as a CIC type as follows:

$$\Sigma A \in \mathsf{Type}.\ \Sigma R \in A \times A \to \mathsf{Type}.\ \forall a \in A.\ R(a,a)$$

In the following, we make use of the projection functions $-^U$, $-^{\mathcal{R}}$ and $-^{\Delta}$ to obtain the first, second and third components of the interpretations of kinds.

We define an interpretation for all kinds by induction on their structure. At base kind, the carrier is simply the type of all sets; relations between A and B are subsets of $A \times B$; and the distinguished identity relation is exactly the equality relation:

$$[\![*]\!] = (\mathsf{Set}, \mathsf{Rel}, \equiv)$$

At higher kinds $\kappa_1 \to \kappa_2$, the interpretation is more involved. The carrier is the set of pairs of functions from the carrier of κ_1 to the carrier of κ_2, along with their associated distinguished identity-preserving identity relations; the relations between two semantic type operators (F, R) and (G, S) are relation transformers; and the distinguished identity relation for (F, R) is just R.

$$\begin{aligned}[\![\kappa_1 \to \kappa_2]\!] = \ & (\{(F,R) \mid F \in [\![\kappa_1]\!]^{\mathcal{U}} \to [\![\kappa_2]\!]^{\mathcal{U}}, R \in (\forall AB.\ [\![\kappa_1]\!]^{\mathcal{R}}(A,B) \to [\![\kappa_2]\!]^{\mathcal{R}}(FA,FB)), \\ & \qquad \forall A \in [\![\kappa_1]\!]^{\mathcal{U}}.\ RAA([\![\kappa_1]\!]^{\Delta}A) = [\![\kappa_2]\!]^{\Delta}(FA)\}, \\ & ((F,R),(G,S)) \mapsto \forall AB.[\![\kappa_1]\!]^{\mathcal{R}}(A,B) \to [\![\kappa_2]\!]^{\mathcal{R}}(FA,GB), \\ & (F,R) \mapsto R\ )\end{aligned}$$

Kinding contexts $\Theta = \alpha_1 : \kappa_1, ..., \alpha_n : \kappa_n$ are interpreted as follows:

$$\begin{aligned}[\![\alpha_1 : \kappa_1, ..., \alpha_n : \kappa_n]\!] = \ & ([\![\kappa_1]\!]^{\mathcal{U}} \times ... \times [\![\kappa_n]\!]^{\mathcal{U}}, \\ & (\theta, \theta') \mapsto [\![\kappa_1]\!]^{\mathcal{R}}(\pi_1\theta, \pi_1\theta') \times ... \times [\![\kappa_n]\!]^{\mathcal{R}}(\pi_n\theta, \pi_n\theta'), \\ & \theta \mapsto ([\![\kappa_1]\!]^{\Delta}(\pi_1\theta), ..., [\![\kappa_n]\!]^{\Delta}(\pi_n\theta)))\end{aligned}$$

We use π_i to denote the ith projection. The carrier component of the interpretation of a kinding context is simply the product of the underlying semantics of the context members, and the relational component is simply the product of the relational components of the context members. This naturally leads to the identity component being defined as the tuple of the identity components of the interpretations of the context members. We overload the projections $-^U$, $-^{\mathcal{R}}$ and $-^{\Delta}$ for the denotations of kinding contexts as well as kinds.

3.5 Interpretation of Types

A kinding judgement $\Theta \vdash A : \kappa$ establishing the well-formedness of a type A is interpreted as a pair of a function $[\![A]\!]^f \in [\![\Theta]\!]^{\mathcal{U}} \to [\![\kappa]\!]^{\mathcal{U}}$ and a relation transformer $[\![A]\!]^r \in (\forall \theta\theta' \in [\![\Theta]\!]^{\mathcal{U}}.[\![\Theta]\!]^{\mathcal{R}}(\theta,\theta') \to [\![\kappa]\!]^{\mathcal{R}}([\![A]\!]^f\theta, [\![A]\!]^f\theta'))$ such that the identity extension property holds. We give the full statement below in Theorem 1.

We now give an interpretation of the kinding judgements of Section 1, by induction on the kinding derivation. Type variables $\Theta \vdash \alpha_i : \kappa_i$, where α_i is the ith variable in Θ, are interpreted using projections:

$$[\![\alpha_i]\!]^f\theta = \pi_i\theta \qquad\qquad [\![\alpha_i]\!]^r\theta\theta'\rho = \pi_i\rho$$

For λ-abstraction of types, we must return a pair of the type operator and the distinguished identity relation on the operator. This is derived from the relational interpretation of the premise and the identity relation on the context. In essence, type-level λ-abstraction in System Fω is interpreted as λ-abstraction at the meta-level:

$$\begin{aligned}
[\![\lambda\alpha : \kappa_1.\ A]\!]^f\theta &= (\lambda X \in [\![\kappa_1]\!]^{\mathcal{U}}.\ [\![A]\!]^f(\theta, X), \\
&\qquad \lambda XY \in [\![\kappa_1]\!]^{\mathcal{U}}, R \in [\![\kappa_1]\!]^{\mathcal{R}}(X,Y).\ [\![A]\!]^r(\theta,X)(\theta,Y)([\![\Theta]\!]^{\Delta}\theta, R)) \\
[\![\lambda\alpha : \kappa_1.\ A]\!]^r\theta\theta'\rho &= \lambda XY \in [\![\kappa_1]\!]^{\mathcal{U}}, R \in [\![\kappa_1]\!]^{\mathcal{R}}(X,Y).\ [\![A]\!]^r(\theta,X)(\theta',Y)(\rho,R)
\end{aligned}$$

Type-level application is interpreted as application at the meta-level:

$$\begin{aligned}
[\![F\ A]\!]^f\theta &= \pi_1([\![F]\!]^f\ \theta)\ ([\![A]\!]^f\ \theta) \\
[\![F\ A]\!]^r\theta\theta'\rho &= [\![F]\!]^r\theta\theta'\rho\ ([\![A]\!]^f\theta)([\![A]\!]^f\theta')([\![A]\!]^r\theta\theta'\rho)
\end{aligned}$$

We now turn to the interpretation of the two constructors of actual types in the kinding system of Section 1. For function types, we use the standard logical relations interpretation of function types as preservation of relations:

$$\begin{aligned}
[\![A \to B]\!]^f\theta &= [\![A]\!]^f\theta \to [\![B]\!]^f\theta \\
[\![A \to B]\!]^r\theta\theta'\rho &= \{(f,g) \mid \forall (x,y) \in [\![A]\!]^r\theta\theta'\rho.\ (fx, gy) \in [\![B]\!]^r\theta\theta'\rho\}
\end{aligned}$$

Finally, a universal type $\forall_\kappa \alpha.A$ is interpreted as the set of all type-parameterised inhabitants of the open type A that are relationally parametric. We use the distinguished identity relation on contexts in this definition, where Θ is the current context in the kinding derivation. The interpretation of universally quantified types as just those elements that are relationally parametric is key to the applications of the model that we present in Section 4. This definition allows us to state results that range over all inhabitants of $[\![\forall_\kappa \alpha.A]\!]^f$, not just those that arise from closed programs.

$$\begin{aligned}
[\![\forall_\kappa \alpha.\ A]\!]^f\theta &= \{x \in (\forall X \in [\![\kappa]\!]^{\mathcal{U}}.\ [\![A]\!]^f(\theta, X)) \mid \\
&\qquad \forall XY, R \in [\![\kappa]\!]^{\mathcal{R}}(X,Y).\ (xX, xY) \in [\![A]\!]^r(\theta,X)(\theta,Y)([\![\Theta]\!]^{\Delta}\theta, R)\} \\
[\![\forall_\kappa \alpha.\ A]\!]^r\theta\theta'\rho &= \{(x,y) \mid \forall XY, R \in [\![\kappa]\!]^{\mathcal{R}}(X,Y).\ (xX, yY) \in [\![A]\!]^r(\theta,X)(\theta',Y)(\rho,R)\}
\end{aligned}$$

The following theorem states that the clauses above are well-defined: all the definitions above are well-typed, and the identity extension property is preserved.

▶ **Theorem 1.** *For all well-kinded types $\Theta \vdash A : \kappa$ there exist functions:*
$[\![A]\!]^f \in [\![\Theta]\!]^{\mathcal{U}} \to [\![\kappa]\!]^{\mathcal{U}}$ *and* $[\![A]\!]^r \in (\forall \theta\theta' \in [\![\Theta]\!]^{\mathcal{U}}.[\![\Theta]\!]^{\mathcal{R}}(\theta,\theta') \to [\![\kappa]\!]^{\mathcal{R}}([\![A]\!]^f\theta, [\![A]\!]^f\theta'))$
such that identity extension holds: $\forall \theta \in [\![\Theta]\!]^{\mathcal{U}}.\ [\![A]\!]^r\theta\theta([\![\Theta]\!]^{\Delta}\theta) = [\![\kappa]\!]^{\Delta}([\![A]\!]^f\theta)$

▶ **Theorem 2** (Soundness of $\beta\eta$)**.** *If $\Theta \vdash A \equiv B : \kappa$ then $[\![A]\!]^f = [\![B]\!]^f$ and $[\![A]\!]^r = [\![B]\!]^r$.*

We extend the interpretation of types to the interpretation of typing contexts in the obvious way: contexts $\Gamma = x_1 : A_1, ..., x_n : A_n$ (where all the A_i are of base kind) are interpreted as tuples of the the interpretations of the individual types, and the relational interpretation is the standard one for logical relations on products.

3.6 Semantics of Terms

We omit the interpretation of terms, which is similar to the interpretation of the terms of System F in a relationally parametric model (see, for example, Bainbridge *et al.* [2]). We just state the key result—the abstraction theorem for System Fω.

▶ **Theorem 3.** *For all well-typed terms $\Theta \mid \Gamma \vdash e : A$ there is a function (the $-^{\mathcal{P}}$ superscript stands for "parametric")*

$$[\![e]\!]^{\mathcal{P}} \in (\forall \theta \in [\![\Theta]\!]^{\mathcal{U}}.\ [\![\Gamma]\!]^f \theta \to [\![A]\!]^f \theta)$$

such that, for all $\theta, \theta' \in [\![\Theta]\!]^{\mathcal{U}}$, $\rho \in [\![\Theta]\!]^{\mathcal{R}}(\theta, \theta')$, $\gamma \in [\![\Gamma]\!]^f \theta$ and $\gamma' \in [\![\Gamma]\!]^f \theta'$,

$$\text{if } (\gamma, \gamma') \in [\![\Gamma]\!]^r \theta\theta'\rho \text{ then } ([\![e]\!]^{\mathcal{P}}\theta\gamma, [\![e]\!]^{\mathcal{P}}\theta'\gamma') \in [\![A]\!]^r \theta\theta'\rho.$$

Moreover, this interpretation is sound for the $\beta\eta$ equational theory of the terms.

The interpretation of terms and the proof that they satisfy the abstraction property must be carried out simultaneously in order to show that the interpretation of Λ-abstraction is well-defined. This is due to the interpretation of the $\forall_\kappa$-types as sets of type indexed values that preserve all relations.

The next theorem relates the model we have defined in this section to the standard non-parametric model in CIC, where we interpret higher kinds simply as functions, and $\forall_\kappa$-types just using the impredicative quantification of the meta-theory. The intention is that this non-parametric semantics represents the "natural" semantics of System Fω in CIC, without the artificial scaffolding of relational parametricity. To state this theorem, it is necessary to extend the calculus with a type of booleans to act as observable values.

▶ **Theorem 4.** *For any closed term $- \mid - \vdash e : \texttt{bool}$, the parametric semantics of Theorem 3 and the non-parametric semantics are equal.*

The proof of this theorem is carried out by the construction of a logical relation between the parametric and non-parametric semantics. The importance of this theorem is that it allows us to use equalities between terms that we prove in the parametric semantics of Theorem 3 to reason about contextual equivalence in the non-parametric semantics. If we prove an equivalence $[\![e_1]\!]^{\mathcal{P}} = [\![e_2]\!]^{\mathcal{P}}$ in the parametric semantics, where e_1 and e_2 may be open terms, then for all contexts $C[-]$ of type `bool`, we have, by the compositionality of $[\![-]\!]^{\mathcal{P}}$, $[\![C[e_1]]\!]^{\mathcal{NP}} = [\![C[e_1]]\!]^{\mathcal{P}} = [\![C[-]]\!]^{\mathcal{P}}[\![e_1]\!]^{\mathcal{P}} = [\![C[-]]\!]^{\mathcal{P}}[\![e_2]\!]^{\mathcal{P}} = [\![C[e_2]]\!]^{\mathcal{P}} = [\![C[e_2]]\!]^{\mathcal{NP}}$, where $[\![-]\!]^{\mathcal{NP}}$ is the non-parametric semantics, and $[\![-]\!]^{\mathcal{P}}$ is the parametric semantics.

4 Applications of Higher Kinded Parametricity

We now demonstrate how, in System Fω, we can define useful data type constructions for *indexed* data types, of kind $\kappa \to *$. Such indexed data types include Generalised Algebraic Data Types (GADTs) [5], by setting $\kappa = *$. We build towards the construction of indexed inductive types in stages, defining equality types, existential types at higher kinds,

product and sum types and then finally indexed inductive types as initial algebras. For each construction, relational parametricity is used to justify the η equality rules.

We reason within CIC, using the model that we constructed in the previous section. When we quantify over all types of a particular kind, we mean to interpret this as all semantic inhabitants of the denotation of this kind. Similarly, when we quantify over terms of a particular type, we mean semantic inhabitants of the denotation of the type. We often omit semantic brackets to reduce clutter. When F is a semantic type of kind κ, we write Δ_F for F's distinguished identity relation $[\![\kappa]\!]^{\Delta}(F) \in [\![\kappa]\!]^{\mathcal{R}}(F, F)$. As a shorthand for the relational interpretations of certain types, we often write type expressions with free type variables replaced by relations.

4.1 Equality Types

It is possible (Jacobs [9], Section 8.1) to extend System Fω with an equality type that records when two types (of arbitrary but equal kinds) are equal. One adds an additional kind indexed family of type operators $\mathsf{Eq}_\kappa : \kappa \to \kappa \to *$ and two kind indexed families of term constants:

$$\mathsf{refl}_\kappa : \forall_\kappa \alpha.\ \mathsf{Eq}_\kappa \alpha\alpha \qquad \mathsf{elimEq}_\kappa : \forall_\kappa \alpha\beta.\ \mathsf{Eq}_\kappa \alpha\beta \to \forall_{\kappa\to\kappa\to *}\rho.(\forall_\kappa\gamma.\ \rho\gamma\gamma) \to \rho\alpha\beta$$

that obey the following β and η equality rules:

$$\frac{A : \kappa \qquad G : \kappa \to \kappa \to * \qquad f : \forall_\kappa \alpha.G\alpha\alpha}{\mathsf{elimEq}_\kappa\ [A]\ [A]\ (\mathsf{refl}\ [A])\ [G]\ f = f\ [A]}\ (\beta)$$

$$\frac{A, B : \kappa \qquad z : \mathsf{Eq}_\kappa AB \qquad G : \kappa \to \kappa \to * \qquad t : \forall_\kappa \alpha\beta.\ \mathsf{Eq}_\kappa\ \alpha\ \beta \to G\alpha\beta}{\mathsf{elimEq}_\kappa\ [A]\ [B]\ z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathsf{refl}_\kappa\ [\gamma])) = t\ [A]\ [B]\ z}\ (\eta)$$

We can define a substitution operation, using elimEq_κ, which will be used in Section 5 below.

$$\begin{aligned} subst_\kappa &: \forall_\kappa \alpha\beta.\mathsf{Eq}_\kappa \alpha\beta \to \forall_{\kappa\to *}\rho.\ \rho\alpha \to \rho\beta \\ subst_\kappa &= \Lambda\alpha\beta.\lambda e.\Lambda\rho.\lambda x.\ \mathsf{elimEq}_\kappa\ [\alpha]\ [\beta]\ e\ [\lambda\gamma_1\gamma_2.\ \rho\gamma_1 \to \rho\gamma_2]\ (\Lambda\gamma.\lambda x.x)\ x \end{aligned}$$

In System Fω, it is possible to implement the equality type. We make the following definitions, encoding the equality type as it own eliminator.

$$\begin{aligned} Eq_\kappa &= \lambda\alpha\beta.\ \forall_{\kappa\to\kappa\to *}\rho.\ (\forall_\kappa\gamma.\ \rho\gamma\gamma) \to \rho\alpha\beta \\ refl_\kappa &= \Lambda\alpha.\ \Lambda\rho.\ \lambda f.\ f\ [\alpha] \\ elimEq_\kappa &= \Lambda\alpha\beta.\ \lambda e.\ \Lambda\rho.\ \lambda f.\ e\ [\rho]\ f \end{aligned}$$

The β equality rule for this implementation of equality types holds in all models of System Fω, just by β reduction. In the parametric model of Section 3, we can also show that the η equality rule holds, by using higher kinded relational parametricity.

▶ **Lemma 5.** *Let A, B be types of kind κ, and G be a type of kind $\kappa \to \kappa \to *$. Then for any $z : Eq_\kappa AB$, and $t : \forall_\kappa\alpha\beta.\ Eq_\kappa\alpha\beta \to G\alpha\beta$, we have (in the model of Section 3):*

$$t\ [A]\ [B]\ (z\ [Eq_\kappa]\ refl_\kappa) = z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (refl_\kappa\ [\gamma]))$$

Proof. Let $\Delta_G = [\![\kappa \to \kappa \to *]\!]^{\Delta}(G)$, the identity-preserving identity relation transformer associated with G. Define another relation transformer $R \in [\![\kappa \to \kappa \to *]\!]^{\mathcal{R}}(Eq_\kappa, G)$ as $R_{AB}S_{A'B'}S' = \{(x, y) \mid (t\ [A]\ [A']\ x, y) \in \Delta_G SS'\}$. We know from the parametricity property derived from z's type that:

$$(z\ [Eq_\kappa], z\ [G]) \in (\forall_\kappa\gamma.\ R\gamma\gamma) \to R\Delta_A\Delta_B \tag{1}$$

where $\Delta_A = [\![\kappa]\!]^{\Delta}(A)$ and Δ_B are the identity relations appropriate to κ for the semantic types A and B. We will instantiate (1) with refl_κ and $\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathit{refl}_\kappa\ [\gamma])$, so we must ensure that these are related at $\forall_\kappa\gamma.\ R\gamma\gamma$. But, for any X, Y and $S \in [\![\kappa]\!]^{\mathcal{R}}(X, Y)$ this reduces to whether $(t\ [X]\ [X]\ (\mathit{refl}_\kappa\ [X]), t\ [Y]\ [Y]\ (\mathit{refl}_\kappa\ [Y])) \in \Delta_G SS$, which follows from t's parametricity property, ensured by its membership of the denotation of a universal type.

Now, $(z\ [Eq_\kappa]\ \mathit{refl}_\kappa, z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathit{refl}_\kappa\ [\gamma]))) \in R\Delta_A\Delta_B$. By the construction of identity relations at higher kinds, $\Delta_G\Delta_A\Delta_B = \Delta_{GAB} = (\equiv)$, so when we unfold the definition of R, we have shown that, in the model, $t\ [A]\ [B]\ (z\ [Eq_\kappa]\ \mathit{refl}_\kappa) = z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathit{refl}_\kappa\ [\gamma]))$. ◀

▶ **Theorem 6.** *The β and η rules stated above for* refl *and* elimEq *hold for the implementations refl and elimEq when interpreted in the model of Section 3.*

Proof. The β equality rule can be seen to hold by expanding the definitions and applying β reduction. Showing that the η equality rule holds requires parametricity. Unfolding and β reducing, the equation to show becomes $z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathit{refl}_\kappa\ [\gamma])) = t\ [A]\ [B]\ z$. From Lemma 5 we know that $z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathit{refl}_\kappa\ [\gamma])) = t\ [A]\ [B]\ (z\ [Eq_\kappa]\ \mathit{refl}_\kappa)$. Now we use Lemma 5 again, with an arbitrary G and $f : \forall_\kappa\gamma.G\gamma\gamma$, setting $t = \Lambda\alpha\beta.\lambda x.x\ [G]\ f$. This gives us $z\ [Eq_\kappa]\ \mathit{refl}_\kappa\ [G]\ f = z\ [G]\ f$, and so by extensionality, $z\ [Eq_\kappa]\ \mathit{refl}_\kappa = z$. Thus we have shown $z\ [G]\ (\Lambda\gamma.\ t\ [\gamma]\ [\gamma]\ (\mathit{refl}_\kappa\ [\gamma])) = t\ [A]\ [B]\ z$, as required. ◀

4.2 Existential Types

As with System F, it is possible to encode existential types in System Fω, only now we have the option of doing so for higher kinds. The specification for existential types goes as follows: for every type $F : \kappa \to *$, there is a type $\exists_\kappa\alpha.\ F\alpha$, and two combinators:

$$\begin{aligned} \mathsf{pack}_\kappa &: \forall_{\kappa\to*}\rho.\ \forall_\kappa\alpha.\ \rho\alpha \to (\exists_\kappa\alpha.\ \rho\alpha) \\ \mathsf{elimEx}_\kappa &: \forall_{\kappa\to*}\rho.\ \forall_*\beta.\ (\forall_\kappa\alpha.\ \rho\alpha \to \beta) \to (\exists_\kappa\alpha.\ \rho\alpha) \to \beta \end{aligned}$$

that obey the following equational rules:

$$\frac{F : \kappa \to *\qquad A : \kappa\qquad x : FA\qquad B : *\qquad f : \forall_\kappa\alpha.\ F\alpha \to B}{\mathsf{elimEx}_\kappa\ [F]\ [B]\ f\ (\mathsf{pack}_\kappa\ [F]\ [A]\ x) = f\ [A]\ x}\ (\beta)$$

$$\frac{F : \kappa \to *\qquad e : \exists_\kappa\alpha.\ F\alpha\qquad B : *\qquad t : (\exists_\kappa\alpha.\ F\alpha) \to B}{\mathsf{elimEx}_\kappa\ [F]\ [B]\ (\Lambda\alpha.\ \lambda x.\ t\ (\mathsf{pack}_\kappa\ [F]\ [\alpha]\ x))\ e = t\ e}\ (\eta)$$

We can implement this specification in System Fω by copying the implementation of existentials in System F. Set $\exists_\kappa\alpha.F\alpha = \forall_*\beta.(\forall_\kappa\alpha.F\alpha \to \beta) \to \beta$ and define

$$\begin{aligned} \mathit{pack}_\kappa &= \Lambda\rho\alpha.\lambda x.\Lambda\beta.\lambda f.f\ [\alpha]\ x \\ \mathit{elimEx}_\kappa &= \Lambda\rho\beta.\lambda f e.\ e\ [\beta]\ f \end{aligned}$$

In the proof of the next lemma, we make use of functional relations. This is a standard concept used in relationally parametric reasoning for System F (for example, Birkedal and Møgelberg [4]). They are called *graph relations* by Hasegawa [8].

▶ **Definition 7** (Functional Relations at Base Kind)**.** For types A, B of kind $*$ and an inhabitant f of the type $A \to B$, we define the relation $\langle f\rangle \in [\![*]\!]^{\mathcal{R}}(A, B)$ as $\langle f\rangle = \{(a, b) \mid fa = b\}$.

▶ **Lemma 8.** *For all $F : \kappa \to *$ and $B : *$, $t : (\exists_\kappa\alpha.F\alpha) \to B$ and $e : \exists_\kappa\alpha.F\alpha$, we have*

$$t\ (e\ [\exists_\kappa\alpha.F\alpha]\ (\mathit{pack}_\kappa\ [F])) = e\ [B]\ (\Lambda\alpha.\lambda x.\ t\ (\mathit{pack}_\kappa\ [F]\ [\alpha]\ x))$$

Proof. By e's parametricity, we know that $(e\ [\exists_\kappa \alpha.F\alpha], e\ [B]) \in (\forall_\kappa \alpha.\Delta_F \alpha \to \langle t \rangle) \to \langle t \rangle$. We will apply this pair to $pack_\kappa\ [F]$ and $\Lambda\alpha.\lambda x.\ t\ (pack_\kappa\ [F]\ [\alpha]\ x)$, so we must show that they are related by $\forall_\kappa \alpha.\Delta_F\alpha \to \langle t \rangle$. Given $X, Y \in [\![\kappa]\!]^{\mathcal{U}}$, $S \in [\![\kappa]\!]^{\mathcal{R}}(X, Y)$ and $(x, y) \in \Delta_F S$, we want to prove $(pack_\kappa\ [F]\ [X]\ x, t\ (pack_\kappa\ [F]\ [Y]\ y)) \in \langle t \rangle$ which is equivalent to $t\ (\Lambda\beta.\lambda f.\ f\ [X]\ x) = t\ (\Lambda\beta.\lambda f.\ f\ [Y]\ y)$, by unfolding the definition of $\langle t \rangle$ and $pack_\kappa$. Now it is possible to prove this equality by using extensionality and f's parametricity.

Thus we have shown $(e\ [\exists_\kappa \alpha.F\alpha]\ (pack_\kappa\ [F]), e\ [B]\ (\Lambda\alpha.\lambda x.t\ (pack_\kappa\ [F]\ [\alpha]\ x))) \in \langle t \rangle$. Unfolding the definition of $\langle t \rangle$, this gives us what we want. ◀

▶ **Theorem 9.** *This implementation of $\exists_\kappa \alpha.F\alpha$, $pack_\kappa$ and $elimEx_\kappa$ satisfies the $\beta\eta$ equality rules for existential types when interpreted in the model of Section 3.*

Proof. The β equality rule follows simply by expanding the definitions and applying the β equality rules of System Fω. For the η rule, we use Lemma 8 to reason as follows:

$$\begin{aligned} elimEx_\kappa\ [F]\ [B]\ (\Lambda\alpha.\lambda x.t\ (pack_\kappa\ [F]\ [\alpha]\ x))\ e &= e\ [B]\ (\Lambda\alpha.\lambda x.t\ (pack_\kappa\ [F]\ [\alpha]\ x)) \\ &= t\ (e\ [\exists_\kappa \alpha.F\alpha]\ (pack_\kappa\ [F])) \end{aligned}$$

The first equality is by unfolding the definition of $elimEx_\kappa$ and the second is by Lemma 8. We now make use of Lemma 8 again, with an arbitrary B and $f : \forall_\kappa \alpha.F\alpha \to B$, setting $t = \lambda x.x\ [B]\ f$. This yields $e\ [\exists_\kappa \alpha.F\alpha]\ (pack_\kappa\ [F])\ [B]\ f = e\ [B]\ f$, and hence, by extensionality, $e\ [\exists_\kappa \alpha.F\alpha]\ (pack_\kappa\ [F]) = e$. Thus we can rewrite the final line in the sequence of equations above to get $t\ e$, as required. ◀

4.3 Categories of Indexed Types

For every kind κ, we define the category of κ indexed types as follows. The objects are types of kind $\kappa \to *$ and morphisms between F and G are inhabitants of the type $\forall_\kappa \gamma.F\gamma \to G\gamma$. We write $F \Rightarrow G$ as shorthand for this type. Identities and composition in these categories are defined as follows:

$$\begin{array}{lcllcl} id_F & : & F \Rightarrow F & \circ & : & (G \Rightarrow H) \to (F \Rightarrow G) \to (F \Rightarrow H) \\ id_F & = & \Lambda\gamma.\lambda x.\ x & \circ & = & \lambda f\ g.\Lambda\gamma.\lambda x.\ f\ [\gamma]\ (g\ [\gamma]\ x) \end{array}$$

The categories of κ indexed types have all finite products and coproducts (sum types). We only sketch the proof here, which is a straightforward extension of the proof for the existence of non-indexed finite products and coproducts in System F.

▶ **Theorem 10.** *The categories of κ indexed types have all finite products and coproducts.*

Proof. *(Sketch)* Pointwise extension of the corresponding constructions for types of base kind in System F [4, 7]. For example, $A \times B = \lambda\alpha.\ \forall_*\beta.\ (A\alpha \to B\alpha \to \beta) \to \beta$. To prove the required universal properties, parametricity is needed. ◀

4.4 Functors and Initial Algebras

For a kind κ, an endofunctor on the category of κ indexed types consists of a pair $(F, fmap_F)$ of a type F of kind $(\kappa \to *) \to (\kappa \to *)$ and an associated function $fmap_F : \forall_{\kappa\to*}\alpha\beta.(\alpha \Rightarrow \beta) \to (F\alpha \Rightarrow F\beta)$ that preserves identities and composition.

We now show that any such functor has an initial algebra, and so we can define κ indexed inductive types. To do so, we need the higher kinded generalisation of the functional relations previously defined at base kind in Definition 7.

▶ **Definition 11** (Functional Relations at Higher Kind). Given types F, G of kind $\kappa \to *$ (i.e., objects of the category of κ indexed types) and f of type $A \Rightarrow B$ define the *functional relation* $\langle f \rangle \in [\![\kappa \to *]\!]^{\mathcal{R}}(A, B)$ as $\langle f \rangle_{XY} S = \{(x, y) \mid (f\ [X]\ x, y) \in \Delta_B S\}$.

▶ **Lemma 12** (Graph Lemma). *For a functor $(F, fmap_F)$ and morphism $f : A \Rightarrow B$, then it is the case that $\Delta_F \langle f \rangle \subseteq \langle fmap_F\ [A]\ [B]\ f \rangle$.*

Proof. Of course, by the inclusion in the lemma statement, we mean that the inclusion holds for all $X, Y \in [\![\kappa]\!]^{\mathcal{U}}$ and $S \in [\![\kappa]\!]^{\mathcal{R}}(X, Y)$. We know by the parametricity property for $fmap_F$ that $(fmap_F\ [A]\ [B], fmap_F\ [B]\ [B]) \in (\langle f \rangle \Rightarrow \Delta_B) \to (\Delta_F \langle f \rangle \Rightarrow \Delta_{FB})$. We proceed by supplying the suitably related arguments $(f, id_B) \in \langle f \rangle \Rightarrow \Delta_B$ and then S, to obtain $(fmap_F\ [A]\ [B]\ f\ X, id_{FBY}) \in \Delta_F \langle f \rangle S \to \Delta_{FB} S$. So, if $(x, y) \in \Delta_F \langle f \rangle S$ then $(fmap_F\ [A]\ [B]\ f\ X\ x, y) \in \Delta_{FB} S$, which implies that $(x, y) \in \langle fmap_F\ [A]\ [B]\ f \rangle S$. ◀

We now show that every endofunctor on the categories of κ indexed types has an initial algebra. This result allows us to define κ indexed inductive types within System Fω and use initiality to reason about them. Following the purely category theoretic definition, we define, for a functor $(F, fmap_F)$, an F-algebra to consist of a pair of a type A of kind $\kappa \to *$ and a morphism $k_A : FA \Rightarrow B$. Given two F-algebras A, k_A and B, k_B, an F-algebra homomorphism between them is a morphism $f : A \Rightarrow B$ such that the equation $k_B \circ fmap_F\ [A]\ [B]\ f = f \circ k_A$ holds. An *initial* F-algebra is an F-algebra $\mu F, in_F$ such that for any other F-algebra A, k_A there exists a unique F-algebra homomorphism $\mu F \Rightarrow A$.

We state the existence of initial F-algebras in type theoretic terms as follows. For any functor $(F, fmap_F)$, we require a type $\mu F : \kappa \to *$, along with two combinators:

$$\begin{aligned} \mathsf{in}_F &: F(\mu F) \Rightarrow F \\ \mathsf{fold}_F &: \forall_{\kappa \to *} \rho.(F\rho \Rightarrow \rho) \to (\mu F \Rightarrow \rho) \end{aligned}$$

that satisfy the following $\beta\eta$ equality rules:

$$\frac{A : \kappa \to * \qquad k_A : FA \Rightarrow A \qquad C : \kappa \qquad e : F(\mu F)C}{\mathsf{fold}_F\ [A]\ k_A\ [C]\ (\mathsf{in}_F\ [C]\ e) = k_A\ [C]\ (fmap_F\ [\mu F]\ [A]\ (\mathsf{fold}_F\ [A]\ k_A)\ [C]\ e)}\ (\beta)$$

$$\frac{A : \kappa \to * \qquad k_A : FA \Rightarrow A \qquad f : \mu F \Rightarrow A \qquad f \text{ is an } F\text{-algebra homomorphism}}{f = \mathsf{fold}_F\ [A]\ k_A}\ (\eta)$$

The β equality rule states that, for any F-algebra A, k_A, $\mathsf{fold}_F\ [A]\ k_A$ is an F-algebra homomorphism. The η equality rule states that $\mathsf{fold}_F\ [A]\ k_A$ is the unique F-algebra homomorphism from $\mu F, \mathsf{in}_F$ to A, k_A.

Within System Fω we can implement the above specification for any functor $(F, fmap_F)$ and show that the β equality rule holds. Moreover, using parametricity we can further show that the η equality rule holds. We define

$$\begin{aligned} \mu F &= \lambda\alpha.\forall_{\kappa \to *}\rho.(F\rho \Rightarrow \rho) \to \rho\alpha \\ fold_F &= \Lambda\rho.\lambda f.\Lambda\alpha.\lambda x.\ x\ [\rho]\ f \\ in_F &= \Lambda\gamma.\lambda x.\Lambda\rho.\lambda f.\ f\ [\gamma]\ (fmap_F\ [\mu F]\ [\rho]\ (fold_F\ [\rho]\ f)\ [\gamma]\ x) \end{aligned}$$

▶ **Lemma 13.** *Let $(F, fmap_F)$ be a functor. Given two F algebras $A : \kappa \to *$, $k_A : FA \Rightarrow A$ and $B : \kappa \to *$, $k_B : FB \Rightarrow B$, and an F-algebra homomorphism $h : A \Rightarrow B$, then for any type C of kind κ and $e : (\mu F)C$, we have $h\ [C]\ (e\ [A]\ k_A) = e\ [B]\ k_B$.*

Proof. By e's parametricity, we know that $(e\ [A], e\ [B]) \in (\Delta_F\langle h\rangle \Rightarrow \langle h\rangle) \to \langle h\rangle\Delta_C$. We will apply this pair to (k_A, k_B), so we must show that $(k_A, k_B) \in \Delta_F\langle h\rangle \Rightarrow \langle h\rangle$. Given $X, Y \in [\![\kappa]\!]^{\mathcal{U}}$, $S \in [\![\kappa]\!]^{\mathcal{R}}(X, Y)$ and $(x, y) \in \Delta_F\langle h\rangle S$, we need to demonstrate that $(k_A\ [X]\ x, k_B\ [Y]\ y) \in \langle h\rangle S$. Unfolding the definition of $\langle h\rangle$, this means we need to show that $(h\ [X]\ (k_A\ [X]\ x), k_B\ [Y]\ y) \in \Delta_B S$. Since h is an F-algebra homomorphism, it suffices to show that

$$(k_B\ [X]\ (\mathit{fmap}\ [A]\ [B]\ h\ [X]\ x), k_B\ [Y]\ y) \in \Delta_B S \tag{2}$$

We know by Lemma 12 that $(x, y) \in \langle \mathit{fmap}\ [A]\ [B]\ h\rangle S$. Unfolding this use of a functional relation, this gives $(\mathit{fmap}\ [A]\ [B]\ h\ [X]\ x, y) \in \Delta_{FB} S$. Now we may use the parametricity property of k_B to show (2).

We now have $(e\ [A]\ k_A, e\ [B]\ k_B) \in \langle h\rangle\Delta_C$, and unfolding the definition of $\langle h\rangle$ yields $(h\ [C]\ (e\ [A]\ k_A), e\ [B]\ k_B) \in \Delta_B\Delta_C$. Since $\Delta_B\Delta_C = \Delta_{BC} = (\equiv)$, we have shown that $h\ [C]\ (e\ [A]\ k_A) = e\ [B]\ k_B$, as required. ◀

▶ **Theorem 14.** *The implementations of μF, in_F and fold_F satisfy the $\beta\eta$ equality rules stated above, in the model of Section 3.*

Proof. The β equality rule is seen to hold by expanding the definitions and β reducing. By itself, this ensures that our definition is a weakly initial algebra. For the η equality rule, we unfold fold_F, and η expand to see that we need to prove $f\ [C]\ e = e\ [A]\ k_A$. By Lemma 13, and that f is an F-algebra homomorphism, we have $f\ [C]\ (e\ [\mu F]\ \mathit{in}) = e\ [A]\ k_A$. We now use Lemma 13 again, with an arbitrary B and $k_B : FB \Rightarrow B$, setting $h = \mathit{fold}_F\ [B]\ k_B$ (which we know to be an F-algebra homomorphism by the β-equality rule). This gives us $e\ [\mu F]\ \mathit{in}_F\ [B]\ k_B = e\ [B]\ k_B$, and so, by extensionality, $e\ [\mu F]\ \mathit{in}_F = e$. Thus we have shown $f\ [C]\ e = e\ [A]\ k_A$, as required. ◀

4.5 Generalised Algebraic Data Types

Given that we can encode existential types, products, coproducts, equality types and initial algebras in parametric System Fω, it is now possible to encode Generalised Algebraic Data Types (GADTs) [5], using the encoding of Johann and Ghani [10] into a system with initial algebras and equality types (the same encoding was used implicitly by Cheney and Hinze [5]). For example, the following Haskell declaration:

```
data Z; data S a
data Vec :: * -> * -> * where
  VNil :: Vec a Z;  VCons :: a -> Vec a n -> Vec a (S n)
```

can be encoded, assuming a kinding context containing $Z : *$, $S : * \to *$ and $\alpha : *$, by the initial algebra of the functor $F\rho\alpha = \mathit{Eq}_*\ \alpha\ Z + (\exists_*\eta.\alpha \times \rho\eta \times \mathit{Eq}_*\ \alpha\ (S\eta))$. We have used the equality type to encode the "transmuting" effect of the constructors on the type parameters.

This encoding is not immediately useful within System Fω because we are not guaranteed anything about equalities between types. Specifically, we cannot be sure that the types Z and $S\eta$ are not equal, so it is not possible to directly translate the following Haskell function:

```
head :: Store a (S n) -> a; head (SCons a _) = a
```

where we know `head` is total because `Z` $\neq$ `S n` for all `n`, by Haskell's generative semantics for `data` declarations. We can simulate this by adding the assumption $\mathit{ZisNotS} : \forall_*\eta.\mathit{Eq}\ Z\ (S\eta) \to 0$ to the context, where $0 = \forall_*\alpha.\alpha$ is the encoding of the terminal object.

This allows us to define the function within System Fω. In the next section we show how to extend the calculus and model with a data kind of natural numbers, which means that we do not have to abuse types to stand for natural numbers.

5 Extension of System Fω with Additional Kinds

In the previous section, we used abstract type constructors Z and S to simulate natural numbers. However, recent versions of the GHC Haskell compiler have been extended with *data kinds*, which lift some inductive data types up to the type level. Yorgey *et al.* [21] provide the details of this lifting. As a step towards modelling data kinds, we demonstrate how our model can be extended with a kind of natural numbers by interpreting them as a discrete "category without composition".

Syntactically, we extend the grammar of kinds with a new kind of natural numbers: $\kappa ::= \ldots \mid \mathtt{nat}$. We also extend the language of types with two new constants and a kind indexed family of constants, with the following kindings:

$$\mathsf{zero} : \mathtt{nat} \qquad \mathsf{succ} : \mathtt{nat} \to \mathtt{nat} \qquad \mathsf{rec}_\kappa : \kappa \to (\mathtt{nat} \to \kappa \to \kappa) \to \mathtt{nat} \to \kappa$$

with the β-equality rules $\mathsf{rec}_\kappa\ A\ B\ \mathsf{zero} \equiv A$ and $\mathsf{rec}_\kappa\ A\ B\ (\mathsf{succ}\ n) \equiv B\ n\ (\mathsf{rec}_\kappa\ A\ B\ n)$.

Semantically, we define $[\![\mathtt{nat}]\!] = (\mathbb{N}, \lambda n_1, n_2.\{* \mid n_1 = n_2\}, \lambda n.*)$, where $\mathbb{N}$ is the set of all natural numbers and the notation $\{* \mid n_1 = n_2\}$ denotes the set $\{*\}$ when $n_1 = n_2$ and the empty set otherwise. Following Hasegawa, and thinking of the interpretation of kinds as categories without composition, the interpretation of $\mathtt{nat}$ is "discrete": there are only relations between equal objects. It is straightforward to define the semantic interpretations of the zero, succ and rec_κ constants.

Given the extension of the calculus and model with this new kind, we can use Theorem 14 to define the inductive type of $\mathtt{nat}$ indexed vectors. We set

$$F\alpha\rho n = (Eq_{\mathtt{nat}}\ n\ \mathsf{zero}) + (\exists_{\mathtt{nat}} n'.\ \alpha \times \rho n' \times Eq_{\mathtt{nat}}\ n\ (\mathsf{succ}\ n'))$$

so that the type of $\mathtt{nat}$ indexed vectors with elements of type A is given by $\mu(FA)$. It is now possible to write the $\mathtt{head}$ function within System Fω without any further assumptions: we can write a term that demonstrates that $\mathsf{succ}\ n$ and zero are not equal, using the *subst* function defined in Section 4.1, and type-level computation using rec_κ:

$$\begin{aligned} \mathit{zeroIsNotSucc} &: \forall_{\mathtt{nat}} n.\ \mathit{Eq}\ \mathsf{zero}\ (\mathsf{succ}\ n) \to 0 \\ \mathit{zeroIsNotSucc} &= \Lambda n.\lambda e.\ \mathit{subst}_{\mathtt{nat}}\ [\mathsf{zero}]\ [\mathsf{succ}\ n]\ e\ [\mathsf{rec}_*\ 1\ (\lambda n\ \alpha.\ 0)]\ * \end{aligned}$$

where $1 = \forall\alpha.\alpha \to \alpha$, the standard encoding of the unit type, and $*$ is the unique inhabitant.

6 Conclusions

We have defined a concrete type theoretic model of relationally parametric System Fω (Theorem 3), based on the idea of interpreting kinds as reflexive graphs, due to Hasegawa and Robinson and Rosolini. This model allows us to reason relationally about the standard non-parametric semantics (Theorem 4). We have investigated some of the consequences of our model, and shown that it is possible to define indexed inductive types within System Fω, with an initiality property that allows for reasoning (Theorem 14). We have also shown that it is possible to extend the model with data kinds, such as the natural numbers.

Acknowledgements Thanks to Patricia Johann, Neil Ghani, Alex Simpson, Thorsten Altenkirch and Lars Birkedal for suggestions and comments on this work. This work was funded by EPSRC grant EP/G068917/1.

References

1. R. Atkey. Syntax For Free: Representing Syntax with Binding using Parametricity. In *Typed Lambda Calculi and Applications (TLCA)*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.
2. E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theor. Comput. Sci.*, 70(1):35–64, 1990.
3. J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2010, pages 345–356, 2010.
4. L. Birkedal and R. E. Møgelberg. Categorical models for Abadi-Plotkin's Logic for Parametricity. *Mathematical Structures in Computer Science*, 15(4):709–772, 2005.
5. J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 90–104, 2002.
6. B. Dunphy and U. S. Redddy. Parametric limits. In *Proc. 19th IEEE Symp. on Logic in Computer Science*, LICS 2004, pages 242–251, 2004.
7. R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, 1994.
8. R. Hasegawa. Relational limits in general polymorphism. *Publications of the Research Institute for Mathematical Sciences*, 30:535–576, 1994.
9. B. Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.
10. P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL 2008, pages 297–308, 2008.
11. M. P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *J. Funct. Program.*, 5(1):1–35, 1995.
12. A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proc. 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2008, pages 423–438, 2008.
13. B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
14. J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *Proc. IFIP 9th World Computer Congress*, volume 83 of *Information Processing*, pages 513–523, 1983.
15. E. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *Proc. 9th Annual IEEE Symp. on Logic in Computer Science*, LICS 1994, pages 364–371, 1994.
16. A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. In *Proc. ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, TLDI 2010, pages 89–102, 2010.
17. I. Takeuti. The theory of parametricity in the lambda cube. Technical Report 1217, Kyoto University, 2001.
18. J. Voigtländer. Free Theorems Involving Type Constructor Classes: Functional Pearl. In *Proc. 14th ACM SIGPLAN International Conference on Functional programming*, ICFP 2009, pages 173–184, 2009.
19. D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(2):175–210, 2010.
20. P. Wadler. Theorems for free! In *Proc. Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA'89, pages 347–359, 1989.
21. B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *Proc. 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, 2012.