Freedom for Proofs!

Representation Independence is More than Parametricity

Irene Yoon

Abstract

Representation independence allows programmers to give different implementations for an abstract interface. Reynolds' characterization of representation independence for System F uses parametricity, and free theorems derived from parametricity give an elegant formulation of how types limit the behavior of functions. This survey paper overviews the recent efforts to effectively bring representation independence results to dependent type theories. The surveyed works demonstrate that extensional notions of equality, through means of extensional equality, univalence, and cubical type theory, can be fruitful for bringing expressive representation independence to dependently-typed programming.

1 Introduction

In Reynolds' seminal paper [27], a fable of Professor Descartes and Bessel illustrates the usefulness of representation independence. Professor Descartes and Bessel had given different definitions for complex numbers, and their seminars were accidentally interchanged. Luckily, this did not matter for the students' understanding because they had explained their mathematical properties at the level of abstraction that encompassed both of their definitions. The moral is that types should enforce levels of abstraction, which leads to the definition of the *abstraction theorem.* The abstraction theorem asserts that parametrically polymorphic functions behave uniformly, which leads to free theorems [36] about such functions. This result is commonly proven by stating the *fundamental theorem* or *fundamental property* of a logical relation, where a logical relation provides a relational interpretation of types.

The use of the abstraction theorem is commonly known as *parametricity*, where parametrically polymorphic functions such as $\forall \alpha . \tau$ will "behave the same" for all possible implementations a for a given α. This leads to a *representation independence* result, where programmers can give different

implementations for the same abstract interface, i.e. the API of a library. Parametricity is also useful in defining practical notions of equality, where one can reason relationally about the behavior of programs. Reynolds-style parametricity (i.e. relational parametricity) has been initially studied for System F, the polymorphic lambda-calculus, and over the past ten years there has been a flourishing body of work in gaining parametricity and representation results to richer systems such as full-spectrum dependent type systems.

Dependent type theories are formal languages where a type can rely on the value of a term. They are commonly enjoyed by programmers through interactive proof assistants, and there are many such proof assistants, including Coq, Agda, Idris, Isabelle/HOL, and Nuprl, which respectively have different underlying type theories. Dependent types offer the power of rich program specifications, where users can formalize expressive safety guarantees and mathematical properties about program behavior. There is a catch: writing mechanized proofs is notoriously labor-intensive, which motivates the need for proof reuse. In an ideal world, programmers would be able to program and prove modulo *equivalences* so that proofs of equivalence of programs can be transported for reuse. Representation independence provides a solution for proof reuse. For a dependently-typed setting, using parametricity also means that values of a given type can be translated to proofs that the values satisfy the relational interpretation. This results in free proofs [11] for dependent type theories, which can be of practical use.

As it will become evident, many have witnessed that this necessitates a behavioral notion of equality which is internal to the theory. In particular, parametricity is wholly compatible with the Calculus of Inductive Constructions (CIC) but cannot be internalized (i.e. it is not a theorem that can be proved within CIC, but a meta-translation can show that it is compatible). Even with the meta-translation step, parametricity alone is not enough for gaining a flexible and easy-to-use form of representation independence results in CIC because of its limited notion of equality. Type theories such as the CIC offer a syntactic notion of equality of terms, which is limited. Most famously, functional extensionality which is consistent with type theories must be added as an axiom for many intensional type theories. A naïve look would suggest that adding such properties axiomatically (See Section 5.5) would solve the problem, but this still does not satisfy the need for an observational equational theory, where an "outside" view of equality can equate syntactically different programs that are behaviorally indistinguishable.

In this survey, we give an overview of the recent efforts of bringing representation independence results to dependent type theories. We will focus

on three settings: (1) Dreyer and Krishnaswami's extensional Calculus of Constructions with internalized parametricity [21], (2) Tabareau et al.'s *univalent parametricity* defined over CIC with axiomatized univalence [29], and (3) Angiuli et al.'s internalized representation independence result which uses univalence and higher inductive types in the de Morgan cubical type theory of Cubical Agda [3].

2 Preliminaries

In this section, we will prime the readers with some background information that will set grounds for basic definitions used.

2.1 Equality Types

There are two methods of defining equality between two terms in a type theory. One method is to define a set of typing judgments to inductively describe the rules of equality, known as *judgmental* equality. This can be used to describe a decidable type-checking algorithm, where coercions between equal types can be silently done. This method of automated coercions is known as *definitional* equality.

A second method is *propositional equality*, where two terms are equivalent with an evidence that the two are equal. Equality types may carry more information than the proposition that two terms are equal. For instance, if we have a proposition $A \vee B$, the proof of this proposition contains the information that either A is true or that B is true. Similarly, the proof of an equality $A = B$ as a type may carry information on how the two types can be equal.

2.2 Intensional vs. Extensional Type Theories

The term "extensionality" is often overused in the literature of type theories. The difference between intensional and extensional type theories comes from how they treat the interplay between judgmental and propositional equality. *Intensional type theories* offer a restrictive view of judgmental equality, where the identity type $\mathsf{Id}_A(x, y)$ is the proof that x and y are equal, and can only implicit coerce between terms with $\alpha/\beta/\eta$ equality. The identity type is characterized by the following rules:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}(a) : \mathsf{Id}_A(a,\, a)}\ \textsc{intro-Id} \qquad \frac{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x,\, y) \vdash C(x, y, z)\ \textit{type} \quad \Gamma \vdash p : \mathsf{Id}_A(a,\, a') \quad \Gamma,\, x : A \vdash c : C(x,\, x,\, \mathsf{refl}(x))}{\Gamma \vdash \mathsf{J}_{x,y,z.C}(p; x.c) : C(a,\, a',\, p)}\ \textsc{elim-J1}$$

$$\frac{}{\Gamma \vdash \mathsf{J}_{x,y,z.C}(\mathsf{refl}(a);\ x.c) = [a/x]c : C(a,\, a,\, \mathsf{refl}(a))}\ \textsc{elim-J2}$$

The elimination form J described by the ELIM and ELIM-EQ rule gives $\mathsf{Id}_A(a,\, a')$ many expected properties of an equality relation, such as symmetry, transitivity, and coercion. An intensional type theory has decidable judgments through requiring explicit coercions for non-$\alpha/\beta/\delta/\eta$ equations.

On the other hand, in an *extensional type theory*, a *proof* of equivalence between two elements x and y, $\mathsf{Eq}_A(x, y)$, can be reflected in the notion of judgmental equality ($\equiv$). There are two ways in which a type theory can be an extensional type theory: definitionally and propositionally.

Definitional extensionality is where we have an "equality reflection rule" of the form (where $\equiv$ is the definitional equality of the respective theory):

$$\frac{\Gamma \vdash a = a' : A}{\Gamma \vdash \mathsf{refl}(a) : \mathsf{Eq}_A(a, a')}\ \textsc{intro-Eq} \qquad \frac{\Gamma \vdash p : \mathsf{Eq}_A(x, y)}{\Gamma \vdash x \equiv y}\ \textsc{elim-Eq}$$

The extensional type theory in Krishnaswami and Dreyer's work that we discuss in Section 4 has definitional extensional equality.

The alternative is to have a *propositionally extensional equality*, where no two terms can be propositionally equal in more than one way. It is common to add axioms to an intensional type theory (such as the vanilla Coq type theory) to gain this property. One option is to add Streicher's *Axiom K* [28], and another is to add uniqueness of identity proofs (UIP) [24]. Axiom K and UIP are equally expressive, i.e. logically equivalent.

"Extensionality" also sometimes refers to *functional extensionality*, which states that two functions are equal iff they output the same results for equal inputs (i.e. $(\forall x\ y\, , x \equiv y \Rightarrow f\ x \equiv g\ y) \Rightarrow f \equiv g$). However, this is an orthogonal definition which does not necessarily coincide with the notion of extensionality we have discussed above.

2.3 Univalence

In common mathematical practice, isomorphic types are treated as the "same", as isomorphic objects often enjoy the same structural properties. Vladimir Voevodsky's *univalence* principle [34] formalizes this practice by stating that isomorphic types are indeed *equal*. More specifically, it states that every equivalence between two types A and B brings about an identity proof $\mathsf{Id}_{\mathcal{U}}(A, B)$. Thus, equal types may be equal in many ways because isomorphisms are *contentful*. Note that Axiom K/UIP implies propositional extensionality, but univalence refutes UIP. Propositional extensionality is also a provable theorem if we have univalence.

Realizing Univalence. There are two main approaches for realizing univalence: one is by adding the principle as an axiom to the type theory, which is commonly used for various Martin-Löf Type Theories (MLTT) and the Calculus of Inductive Constructions. While seemingly convenient, axioms do not have computational content and lead to *stuck* terms, i.e. type theory with axioms results in a programming language where closed terms of a natural number type cannot reduce to a numeral expression. More precisely, the property where all closed expressions reduce to their canonical form representation is known as *computational adequacy*, where axiomatized theories lack computational adequacy.

Another is the constructive account of realizing univalence through cubical type theories [15], which gives computational content to univalence. Cubical type theories allow full expressivity of using univalence, such as gaining computational content for functional extensionality. The techniques discussed in this survey, Tabareau et al.'s (See Section 5) *univalent parametricity* and Angiuli et al.'s (See Section 6) use of the *structure identity principle* (SIP), is agnostic to the method of how univalence is realized.

3 Parametric Type Theories

Reynolds' relational parametricity for System F captures the uniform behavior of polymorphic functions. However, Reynolds later showed that this theory can only be formalized in a meta-theory with an impredicative universe, such as the Calculus of Inductive Constructions. Parametric type theories are an exploration of how to formalize parametricity in such dependently-typed setting. The initial efforts to introduce parametricity into a dependently-typed setting were through using syntactic logical relations. Starting from the parametricity proof for System F by Girard [19], Johann and Voigtländer [20] transported the approach to the calculus extended with explicit strict-

ness, then Vytiniotis and Weirich [35] to F_ω extended with representation types, Takeuti [30] to the λ-cube, and Neis et al. [23] to a calculus with dynamic casting.

Bernardy et al. showed through a series of work [10, 12, 9] that *pure type systems* [6], which classify simple dependent type systems in the style of Barendregt, respect the parametricity theorem but are not able to internalize the theorem and then show subsequent extensions to the systems that will allow for the internalization of parametricity. A surprising artifact is a demonstration by Bernardy et al. [10] that the syntax of CIC is compatible with parametricity but it cannot internalize the theorem, i.e. the abstraction theorem cannot be stated and proved *within* the type theory and thus requires a meta-theoretic translation step. [1]

$$
\begin{aligned}
[\![\mathsf{Type}_i]\!]\ A\ B &\triangleq A \to B \to \mathsf{Type}_i \\
[\![\Pi(a:A).B]\!]\ f\ g &\triangleq \Pi(a:A)(a':A')(a^\epsilon : [\![A]\!]\ a\ a').[\![B]\!]\ (f\ a)\ (g\ a') \\
[\![x]\!] &\triangleq x^\epsilon \\
[\![\lambda(x:A).t]\!] &\triangleq \lambda(x:A)(x':A')(x^\epsilon : [\![A]\!]\ x\ x').[\![t]\!] \\
[\![t\ u]\!] &\triangleq [\![t]\!]\ u\ u'[\![u]\!] \\
[\![\cdot]\!] &\triangleq \cdot \\
[\![\Gamma,(x:A)]\!] &\triangleq [\![\Gamma]\!]\ ,(x:A),(x':A'),(x^\epsilon : [\![A]\!]\ x\ x')
\end{aligned}
$$

Figure 1: Parametricity translation for CC_ω by Bernardy et al. [12]

To overview the syntactic approach, the Bernardy approach goes as follows. In order to set grounds for the parametricity translation in Section 5, we focus on CC_ω, the Calculus of Constructions with predicative universes [2]. The syntax has a hierarchy of universes Type_i, variables, applications, lambda expressions, and dependent function types.

$$A, B, M, N ::= \mathsf{Type}_i \mid x \mid M\ N \mid \lambda(x:A).M \mid \Pi(x:A).B$$

Showing parametricity results occur through defining a relational interpretation of types. A logical relation of a type A is given as $[\![A]\!]$, which can

[1] In particular, the issue in internalizing parametricity is that giving such a parametric interpretation does not preserves types, and thus breaks subject reduction [22].

[2] Adding an impredicative universe has little impact, and thus CC_ω is used for ease of presentation.

relate two terms at the type. $[\![A]\!]\ a_1\ a_2$ states that terms a_1 and a_2 are related at type A. Figure 1 describes the parametricity translation for CC_ω. The translation for universe Type_i is defined as binary relations of types. The prime notation ($'$) denotes duplication with renaming, where each free variable x is replaced by x'. The dependent function type $\Pi a : A.B$ translation states that related inputs at A lead to related outputs at A as witnessed by e. x^ϵ is the witness that free variables give related outputs from the translation. The function type $\lambda x : A.t$ translation is a function that takes two arguments and the witness that the two are related. The application $t\ u$ translation applies the translation of t to the original argument and its renamed duplicate along with the translation of u. The type environment translation gives a duplicate renaming for each variable and the relational witness x^ϵ.

With this translation, a Reynolds-style parametricity theorem can be proven, stated as the following as a meta-theorem:

Theorem 1 (Abstraction Theorem)**.** *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!] \vdash [\![t]\!] : [\![A]\!]\ t\ t'$.

This means that if t has type A under context Γ, then related interpretations of t are related by A in related environments Γ.

The Abstraction Theorem follows a standard logical relation result of showing the fundamental property. If the abstraction theorem can be stated and proved within the type theory, it is said to have *internalized parametricity*. This is as opposed to an externalized parametricity, where the theorem is stated through a meta-theoretic translation.

There is a limit to this approach: in an intensional type theory, one can use the abstraction theorem to prove propositionally that two programs are equivalent to each other. However, propositional equality is not reflected onto definitional equality in an intensional theory. This means that the coercion between these two programs are not automatic, and is a practical issue when wanting automated transport between proofs about equivalent terms. Section 5 will describe how to restrict the notion of parametricity with univalence for automated proof transport.

Another issue in integrating parametricity theory to dependent type theories is that parametricity in type theory (in its simple form) does not admit an identity extension lemma. The identity extension lemma is crucial for proving theorems involving equality—it ensures that if an identity type is passed as relations for the arguments of a type constructor, the resulting relation is equivalent to the identity. This has been addressed by considering the reflexive graph model on small types [5], and then with an extension of type theory with a parametric function type [25].

4 Parametricity and Realizability Semantics

Bernardy et al. [10] show that for a dependently-typed pure type system (PTS) every term in the type theory respects parametricity results, but that there is no way to internalize this fact. The goal of internalized parametricity to allow the already compatible parametricity theorem to be provable within the type theory. One method for this is a purely syntactic approach: a syntactic extension of the system [12] with *operators* appealing to parametricity can internalize parametricity. Specifically, theorem 1 (Abstraction Theorem) gives for each variable $t : A$ in Γ, an explicit witness that t is parametric in the environment is needed ($[\![t]\!] : [\![A]\!]\ t\ t'$). The operator $\lceil\!\lceil x \rceil\!\rceil$ denotes the witness that x satisfies the parametricity condition on its type. Essentially, the following typing rule which expresses that if x is found in the context, then it is valid to use $\lceil\!\lceil x \rceil\!\rceil$ is added to the type theory.

$$\textsc{param}\quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash \lceil\!\lceil x \rceil\!\rceil : x \in [\![A]\!]}$$

However, this approach is limited in that it only applies to closed terms and that it cannot be used to internalize program equivalences as equalities.

Krishnaswami and Dreyer give a relationally parametric model for an extensional calculus of constructions through a realizability-style interpretation of types, as described in the following section. What this offers is the ability to validate strong equality axioms and induction principles using parametricity. The validated axioms can then be soundly added into the type theory (i.e. internalized), which is justified by parametric reasoning. The main technical innovation in this work is the use of *quasi-PER*s for a heterogeneous representation of equality. This gives a more faithful representation independence result because it allows the theory to relate representations of data that may have different type representations (e.g. Peano nats and Church numerals).

4.1 BHK Interpretation and Realizability Semantics

> If you want to convince yourself of the fact that it is raining, there is no other way than to expose yourself directly to the falling rain.
>
> — Per Martin-Löf

The Brouwer–Heyting–Kolmogorov interpretation (BHK interpretation), well-known for its proposition-as-types mantra is one of the most profound insights in type theory. A pleasant result of this interpretation is the correspondence of a derivable type of a program to a logical proposition. At a cursory glance, this looks similar to the Curry-Howard correspondence, but a closer look reveals a deeper notion. When we pay closer attention, there is a proposition-, or truth-, forward insight: true propositions, that may not already be part of the set of syntactic rules of a program (e.g. through its pre-defined typing rules), should be *realizable* as valid types of a program. This understanding of what types are meant to be (as truthful propositions of programs) is what lies at the heart of a *realizability semantics*.

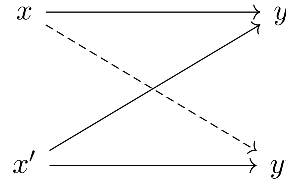
In a realizability semantics, every type of a program has a semantic interpretation given by its *realizer*, corresponding to a valid proposition according to the program execution. We can, on the converse, construct a semantically valid realizer for a program to extend the set of possibly incomplete typing rules of a program. The power of parametricity allows us to gain *free theorems* of a program and add rules in a dependently-typed setting which would otherwise have not been possible to prove in the type theory. Through the realizability semantics that Krishnaswami and Dreyer present, consequences that are derivable from parametricity can be added as sound axioms. Note that adding an axiom using this method would possess computational content (corresponding to the realizer), where the realizer is the untyped term inhabiting the semantic interpretation of that axiom's type.

Their realizability-style model interprets types as relations through *logical relations*. The standard result for the logical relations technique, commonly known as the *fundamental theorem* or *fundamental property*, shows only a reflexive congruence for well-typed terms and does not provide a model of *equality*, which should be symmetric and transitive. Typically, the relational interpretation of types given by the logical relation is required to be a partial equivalence relation (PER), a symmetric and transitive relation. There is trouble in how we mandate symmetry in the relations because by definition

symmetry requires a relation to be homogeneous and thus cannot relate unequal types that may be representationally equivalent.

4.2 Quasi-PERs

The need for a PER representation that is heterogeneous is solved through the use of *quasi-PER*s (a.k.a. *difunctional* or *zigzag-complete* relations) as the model of types. As we'll see, it generalizes PERs to the asymmetric case in a convenient way.



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

The zigzag-completeness is best understood pictorially, with the diagram above. A QPER $R \subseteq X \times Y$ induces a PER on X with $R \circ R^{-1}$, and a PER on Y with $R^{-1} \circ R$.

Thanks to its asymmetry, QPERs can relate terms of different types while also having a canonical equivalence relation. Dreyer and Krishnaswami define their canonical equivalence on *pairs of related terms* on a QPER, where this induced paired equivalence is a PER.

Definition 2 (Canonically induced PER)**.** *Every QPER $Q \subseteq R \times S$ induces an equivalence relation $\sim_Q \subseteq Q \times Q$ (and hence a PER on $R \times S$), defined as $(a_1, a_2) \sim_Q (b_1, b_2)$ iff the zigzag $\{(a_1, a_2), (b_1, b_2), (a_1, b_2), (b_1, a_2) \subseteq Q\}$.*

Quasi-PERs give a significantly coarser notion of equality due to their inherent notion of asymmetry. It can give a single relational model supporting symmetry and transitivity while being able to relate terms with different type representations. This is an advancement to prior approaches, which require building a PER model of types *as well as* a relational model between such PERs.

4.3 Extensional Calculus of Constructions

The calculus at hand is an explicitly-typed extensional calculus of constructions with an identity type and an elimination rule for equality based on equality reflection. Equality reflection is supported: if $\Gamma \vdash e_p : \mathsf{Eq}_x(e, e')$, then $\Gamma \vdash e \equiv e' : X$. The definitional equality is $\beta\eta-$theory of lambda calculus, plus the equality types in their calculus enjoy uniqueness of identity proofs (UIP). An extensional CoC is used in this work because reflection

allows the use of equality axioms (such as η equality for Church encoding of pairs) that are computationally adequate/well-behaved, as opposed to an intensional system where equality axioms make subject reduction fail, resulting in *stuck* terms.

4.4 Semantic Interpretation and Fundamental Property

The interpretation of contexts $[\![\Gamma\ \mathsf{ok}]\!]$, where $\Gamma\ \mathsf{ok}$ are well-formed contexts, is the set of *grounding environments* γ which satisfy it. There are no syntactic well-formedness constraints on the interpretation because the operational semantics does not examine the types of the term, as the relation on terms will contain the necessary semantic constraints. The notion of equivalence on environments forms a PER.

$$[\![\Gamma \vdash * : \mathsf{kind}]\!]\ \gamma \triangleq \left\{ R \in \textsc{Qper}(\textsc{Exp},\ \textsc{Exp}) \;\middle|\; \begin{array}{c} \forall (e_1,\, e_2) \in R.\ e_1 \downarrow \wedge\ e_2 \downarrow \wedge \\ \forall (e_1',\, e_2') \in R, (e_1', e_2') \in \textsc{Exp}^2. \\ e_1 \leftrightarrow^* e_1' \wedge e_2 \leftrightarrow^* e_2' \\ \implies (e_1', e_2') \in R \end{array} \right\}$$

Figure 2: Candidate relations, which form the semantic interpretation of base kinds.

The interpretation of kinds happens in two steps: first, a *pre-interpretation* is given as an approximate interpretation, and then a main interpretation relative to a context γ is given. The main interpretation of the base kind, $[\![\Gamma \vdash * : \mathsf{kind}]\!]$ is given in Figure 2, where it is a $\textsc{Qper}$ of terminating terms that is closed under expansion and reduction.

The interpretation of types is mostly unsurprising, where lambda-abstraction and types of a base kind follow the usual rule for function types in logical relations. The identity type is interpreted by a relation containing at most one pair of values, which validates Axiom K.

The fundamental property requires that the environments γ are well-formed, and then all well-typed terms are self-related by the corresponding relational interpretation of types. This theorem justifies the use of parametricity and implies consistency such that only inhabited relational interpretations have a syntactically inhabited type. There are many judgment forms in the model, and thus the property has clauses for each judgment form. For instance, the fundamental property base kinds is as following:

Theorem 2 (Fundamental Property for Kinds)**.** *Suppose* Γ ok*, and* $\gamma, \gamma' \in [\![\Gamma\ \mathsf{ok}]\!]$ *such that* $\gamma \sim \gamma'$*. Then, (1) If* $D :: \Gamma \vdash \kappa : \mathsf{kind}$*, then* $[\![D]\!]\ \gamma = [\![D]\!]\ \gamma'$*, and (2) If* $\Gamma \vdash \kappa \equiv \kappa' : \mathsf{kind}$*, then* $[\![D]\!]\ \gamma \in [\![\Gamma \vdash \kappa : \mathsf{kind}]\!]\ \gamma$*.*

4.5 Free Theorems and More

In this setting, dependent records, the induction principle for natural numbers, existential types, and quotient types can be internalized within the extensional calculus of constructions. Details on how to provide realizers for each construction are provided in [21], and this section presents the realizers for dependent records.

Dependent Records. While the cartesian product can be defined easily in CoC, dependent records (Σ-types) are not and are realizable in this model.

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

and its introduction form:

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

This corresponds to a *weak* pair type, because its eliminator form is $\mathsf{let}\ (x, y) = p\ \mathsf{in}\ e'$ instead of projective eliminators such as $\pi_i(p)$. With parametricity, one can realize *strong* eliminators [17] with the following realizers, which are semantically well-typed but syntactically ill-typed:

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

The projective second eliminator snd is not syntactically well-typed. Instead, it has the correct *semantic* type and thus is guaranteed to have good computational behavior and can be soundly added as an axiom to the system. The proof for semantic well-typedness for snd is direct, and relies on the well-formedness of the contexts in order to appeal to the fundamental property.

Church numerals can be defined with polymorphism in CoC, but there is no syntactically typable term for the induction principle for Church numerals. Namely, there is no syntactically typable term of:

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

Surprisingly, the realized dependent records can be used to prove the semantic well-typedness of this term. Specifically, the following term is related to itself at the above type.

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

Using a dependent pair, the two arguments of $\Pi n : \mathbb{N}.\ P(n) \to (P\ (S\ n))$ can be packaged into a single argument, which is the expected argument for the step function of the Church encoding. Then, parametricity can be used to prove that for all n, applying the iterator $n\ (\Sigma x : \mathbb{N}.\ P(x))\ o\ h$ results in a record where its first component is n, and its second component is of type $P(n)$. More details of the proof are available in the appendix of [21].

5 Parametricity and Univalence

A practical result from having parametricity in a dependent type setting is its ability to transport proofs between equivalent representations. Tabareau et al. [29] show that parametricity alone may not be enough for automated proof transport, and make the connection of its usage to *univalence*. Univalence, a principle proposed by Voevodsky [33], concretizes the mathematical intuition that isomorphic types should be treated as the "same" type since isomorphic objects enjoy the same structural properties. Univalence and parametricity are similar in that they can both be used to transport proofs across isomorphic types.

The limits of parametricity are as follows. First is the problem of using parametricity in an intensional type theory, where two parametrically related functions are propositionally equal to each other. Because an intensional type theory does not reflect propositional equality to definitional equality (which offers silent coercions between equal types), this is problematic for showing automatic coercions between parametrically equivalent programs. This issue is solved by limiting the parametric translation to relations that respect *univalence*, which is called *univalent parametricity*.

Second, parametricity requires the user to define a common interface *a priori* (e.g. a module signature for natural numbers needs to be defined in advance before giving it a Church numeral representation and a binary representation which are interchangeable). Cohen et al. claim that stating an abstract interface a priori can be difficult engineering-wise [16], which Tabareau et al. state as the *anticipation problem* of parametricity. Typically,

a polymorphic interface requires a common abstract type α, where various implementations $\sigma : \alpha$ can be given (this is the notion of a *homogeneous parametricity*).

Tabareau et al. suggest a *heterogeneous parametricity* which has two concrete types α and α', where given two implementations $\sigma : \alpha$ and $\sigma : \alpha'$, one can demonstrate the parametric relation between these different concrete types *directly*, without having a common abstract type. Because a common abstract type between implementations is not necessary in order to show parametricity results, the anticipation problem is solved. In addition to solving the engineering concern that is posed, this approach gains additional expressivity because concrete types enjoy many definitional equalities that abstract types do not. For instance, if one parameterizes proofs by an implementation of $\mathbb{N}$ with $+$, the abstract $+$ will not reduce on any input, whereas a concrete implementation will reduce on 0.

In short, this results in a *univalent parametricity* that can relate terms at different types and can automatically coerce between parametrically equivalent terms and provide proof transport. The framework presented by Tabareau et al. emphasizes the practical use of univalent parametricity, implementing an automated tool for proof transport in Coq. This work has two main contributions: (1) *univalent parametricity*, which combines parametricity with axiomatic univalence while preserving computational content, and (2) a solution to the *anticipation problem*. The goal is to automatically transport representationally independent, but equivalent, data. Parametric transport is aware of the syntactic structure of the term and thus can do structural rewriting of observationally equivalent terms. Univalent transport, on the other hand, induces a proof obligation where the user must prove the isomorphism between two programs, but is not aware of the type structure of the given term. Univalent parametricity is the amalgam between these ideas, where user-provided equivalences are used when possible while solving the computational problem for parametric transport.

5.1 Heterogeneous Parametricity

Parametricity results typically suffer from the *anticipation problem*, where a common interface needs to be defined *a priori* for parametricity to take effect. For instance, if a user would like to relate $\mathbb{N}$ to a binary representation of numbers, they must rely on a common interface that captures their algebraic structure. This is largely because the *fundamental property* result from logical relations is usually a reflexive homogeneous relation. When relying on a homogeneous instance, one needs to relate the type of naturals to the

type of binary numbers at the interpretation of Type at an appropriate universe level (i.e. $[\![\mathsf{Type}_i]\!]\ \mathbb{N}\ \mathsf{Bin}$). However, a *heterogeneous* parametricity can relate inhabitants of these different, but related, types to each other *directly*. For instance, we can use parametricity to relate the inhabitants $\mathsf{0}$ and $\mathsf{0}_{\mathsf{Bin}}$ which are at different types.

5.2 Limits of Parametricity in an Intensional Type Theory

The *computation problem* arises from using parametricity in an intensional type theory. The type theory is kept intensional in this setting to keep type-checking decidable. Parametricity in this setting relates the behavior of functions propositionally, but this does not imply that the functions are definitionally equal. Thus, parametricity does not scale to computation at the type level, since the proof of parametricity between types cannot be used for their definitional coercion (whereas in an extensional theory, one can use equality reflection to use parametricity results for type-level coercions).

5.3 Univalence to the Rescue

To gain back definitional equality for parametrically related types, we can use univalence. With univalence, which is added axiomatically in this setting (we'll return to this later), propositional type equalities are type equalities, which means that there are *computationally relevant* transports between isomorphic types.

A function $f : A \to B$ is an *equivalence* iff there exists a function $g : B \to A$ paired with proofs that f and g are inverses of each other. The *section* property states that $\forall a : A, g(f(a)) = a$ and the dual *retraction* property states that $\forall b : B, f(g(b)) = b$.[3] Additionally, the equivalence is uniquely determined by the function f.

Definition 3 (Type Equivalence)**.** *Two types A and B are isomorphic to each other iff there exists a function $f : A \to B$ that is an equivalence.*

The isomorphic functions, also known as *transport functions*, construct terms of one type to the other, which can be provided as evidence for showing judgmental equality between two types. The transport functions must be provided manually, i.e. it will generate a proof obligation for the user,

[3]For expository purposes, regard the notion of equality ($=$) used here as propositional equality. More precisely, the definition uses a notion of a *path* that indicates how two types are equal. A detailed explanation of the univalence principle can be found in the HoTT Book [31].

but solves the *computation problem* with parametricity in an intrinsic type theory.

Univalence concretizes the intuition that isomorphic types are propositionally equal. More precisely:

Definition 4 (Univalence)**.** *For any two types A, B, the canonical map* $\mathsf{Id}(A, B) \to (A \simeq B)$ *is an equivalence.*

Univalence makes the promise of immediate transport, where if A and B are equivalent, any $P\ A$ can be converted to an equivalent $P\ B$. Since there is no syntactic restriction on the types or the transport functions, what this gives is a *black-box transport* which does not care about the syntactic structure of the term and type that are being related to.

5.4 Limits of Univalence for Automatic Transport

For automatic transport, univalence needs to be extended into a heterogeneous setting. This is a problem for automation, because every time that two representations with different types need to be proven equal, the coercion requires an extra generalization step. This generalization step can become quickly complicated and does not scale to automation so programmers must explicitly provide evidence of a transport function.

In addition, because univalence is axiomatized in this setting, it is necessary to provide additional information during coercion because the use of axioms leads to stuck terms that do not reduce to their canonical forms. To get around this, a use of typeclasses that mimic computational rules in cubical type theory allows for proof transport that retains computational content. The details of the typeclass implementation of the tool are elided here and can be found in Tabareau et al.'s report[29].

5.5 Univalent Parametricity

The goal of univalent parametricity is to solve all of these above problems: (1) solve the anticipation problem of homogeneous parametricity, (2) solve the computational problem of using parametricity in an intrinsic type theory, and (3) use univalence only when desired, as it solves the computational problem but adds additional programmer burden to provide the transport evidence. This idea, transport á la carte, refine automatic transport by establishing additional univalent relations. We retrieve a parametricity property that can relate terms of different types when the global environment provides evidence that they are in univalent relation. Univalent parametricity uses

parametricity to infer new univalent relations from existing ones, and then exploit induced equivalences to transport proofs and terms. When types A and B are in univalent relation, any *open term* $t : A$ is in univalent relation to its transport at type B.

Take Your Equivalence with You. [4] Univalent parametricity takes equivalences into account when the types are translated. The translation for a universe, $[\![\mathsf{Type}_i]\!]$, is given by a relation $R : A \to B \to \mathsf{Type}_i$ (just as Bernardy did for their parametric translation), which is fortified with an equivalence $e : A \simeq B$ and a *coherence condition* for the equivalence w.r.t. the relation. Coherence states that the relation coincides with propositional equality up to the equivalence. More precisely, for all $a : A$ and $b : B$ for types A and B, $R\ a\ b \simeq (a = \uparrow_e b)$. The up-to-equivalence is formulated with the transport defined on the equivalence ($\uparrow_e$).

The translation at the *term position*, $[T]$, must be distinguished from a translation at the *type position*, $[\![T]\!]$. Translating Type_i at the type position is what we have discussed so far:

$$[\![\mathsf{Type}_i]\!]\ A\ B \triangleq \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).\, \Pi a\ b.\, (R\ a\ b) \simeq (a = \uparrow_e b)$$

Types in CIC can only be "observed" through being in type position (i.e. inhabited), so translations in a term position can collect more information. In this instance, when translated at the term position (i.e. left of the ":"), we have a triple which provides the proofs for the information about type isomorphisms we carry in our type translation. The first line of Figure 3 shows the term translation, where $\mathsf{id}_{\mathsf{Type}_i}$ is the identity equivalence on the universe, and $\mathsf{univ}_{\mathsf{Type}_i} : \Pi A\ B. [\![\mathsf{Type}_i]\!]\ A\ B \simeq (A = B)$, a proof that the univalent relation in the universe is coherent with equality on the universe.

Full univalent parametricity translation is shown in Figure 3. The translation targets CIC_u, CIC augmented with the univalence axiom. In addition to $[\![A]\!]$, we need the equivalence $[\![A]\!]^{eq}$ and witness for coherence $[\![A]\!]^{coh}$. The abstraction theorem is as follows, where $\Gamma \vdash_u t : T$ states that the term is typable in CIC_u:

[4]Notice that this is an explicit, by-hand simulation of univalence. Compare this with an extensional type theory, where it is not necessary to provide such an equivalence and coherence condition explicitly. More precisely, in an extensional type theory does not induce the problem that propositional equality as shown by parametricity cannot be reflected in judgmental equality. Thus, this approach of univalent parametricity is similar to introducing setoids in an intensional type theory where a type carries an equivalence relation. Instead, as we'll see here, the relational interpretation of a type carries an equivalence relation.

Theorem 3 (Abstraction Theorem). *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!] \vdash_u [t] : [\![A]\!]\ t\ t'$.

Heterogeneous univalent parametricity. The methodology behind heterogeneous parametricity can be used for univalent parametricity. A global context Ξ, stated as a telescope Ξ_n, allows the fundamental property to be extended heterogeneously. The context Ξ is defined as a constant triple, where each triple consists of two constants c° and $c^\bullet$ (such as zero (0) in Church numeral representation, and zero (0_{Bin}) in binary number representation), and a witness $c^\otimes$ that the two constants are parametrically related. The telescope Ξ_n defining the context is as follows:

$$\begin{aligned}
&\Xi_0 = \cdot \\
&\Xi_1 = (c_1^\circ : A_1^\circ; c_1^\bullet : A_1^\bullet; c_1^\otimes : [A_1]_u^{\Xi_0}\ c_1^\circ\ c_1^\bullet) \\
&\ldots \\
&\Xi_n = \Xi_{n-1}, (c_n^\circ : A_n^\circ; c_n^\bullet : A_n^\bullet; c_n^\otimes : [A_n]_u^{\Xi_{n-1}}\ c_n^\circ\ c_n^\bullet)
\end{aligned}$$

The definition on Figure 3 is extended for constants as:

$$[c^\circ]^\Xi ::= c^\otimes \text{ when } (c^\circ : _; c^\bullet : _; c^\otimes : _) \in \Xi$$

As a result, with univalent parametricity, we retrieve both (1) *heterogeneous parametricity property* and a (2) *univalent property*[5] which is oblivious to the structure of the term, and states that if two types A and B are univalently related, there is a canonical term in B related to a transported term t of type A. In their development, univalent property is internal to the theory, while the parametricity property is not. Thus, the univalent property may be used in any context on open terms, which fits their purpose of defining automatic transport.

6 Representation Independence in Cubical Type Theory

The motivation behind cubical type theory is to have a theory of univalence with computational content. Equality in the theory will have "computational" information about how the two objects in question are equal. Intuitively, this information is necessary and available because two objects can be isomorphic in many ways. As a consequence, univalence will be derivable

[5] In the paper, the authors state these as "white box fundamental property", and "black box fundamental property", respectively.

$$
\begin{aligned}
[\mathsf{Type}_i] &\triangleq \lambda(A\ B : \mathsf{Type}_i), \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).\\
&\quad \Pi ab.(R\ a\ b) \simeq (a =\uparrow_e b); id_{\mathsf{Type}_i}; \mathsf{univ}_{\mathsf{Type}_i}\\
[\Pi(a : A).B] &\triangleq \lambda(f : \Pi(a : A).B)(g : \Pi(a' : A').B).\\
&\qquad \Pi(a : A)(a' : A')(a^\epsilon : [\![A]\!]\ a\ a').[\![B]\!]\ (f\ a)\ (g\ a');\\
&\quad \mathrm{Equiv}_\Pi A\ A'[A]\ B\ B'[B]; \mathsf{univ}_\Pi A\ A'[A] B\ B'[B]\\
[x] &\triangleq x^\epsilon\\
[\lambda(x : A).t] &\triangleq \lambda(x : A)(x' : A')(x^\epsilon : [\![A]\!]\ x\ x').[\![t]\!]\\
[t\ u] &\triangleq [\![t]\!]\ u\ u'[\![u]\!]\\
[\![A]\!] &\triangleq [A].1 \qquad [\![A]\!]^{eq} \triangleq [A].2 \qquad [\![A]\!]^{coh} \triangleq [A].3\\
[\![\cdot]\!] &\triangleq \cdot\\
[\![\Gamma, (x : A)]\!] &\triangleq [\![\Gamma]\!], (x : A), (x' : A'), (x^\epsilon : [\![A]\!]\ x\ x')
\end{aligned}
$$

Figure 3: Univalent Parametricity translation for CC_ω by Tabareau et al. The blue text indicates the *univalent* information added to the translation (See Figure 1).

inside the theory, without needing to be axiomatized. This is as opposed to Homotopy Type Theory (HoTT) [31], which axiomatizes univalence. The use of axioms results in *stuck* terms that are not able to reduce. For instance, in Tabareau et al.'s work (See Section 5), the tool must painstakingly maneuver coercions between typeclasses that simulate computational rules that are at the foot of cubical type theory. Even worse, stuck terms lack *computational adequacy*, the property that all closed terms of natural number type compute numerals.

In this section, we will focus on how a principle derived from univalence alone can be used to bring representation independence results. More specifically, we would like to have representation independent results that are internal to the theory, without having to rely on the use of axioms. Thus, we focus on a cubical type theory, which gives a computational interpretation to the univalence principle. Angiuli et al. [3] use the *structure identity principle* in a cubical type theory to obtain equality of implementations. There have been other noteworthy expeditions to bringing parametricity, internal and external, to the cubical type theory. For cubical type theory, Cavallo and Harper [14]'s parametric cubical type theory, and Nuyts et al.'s internal

parametricity [25] take inspiration from Bernardy et al.'s presheaf model [8] for internal parametricity.

6.1 Crash Overview of Cubical Type Theory

In this overview, we provide a brief sketch of the key ideas in cubical type theory. Specifically, this will present the view of path types from a de Morgan cubical type theory, following Cohen et al.'s system [15]. This is also commonly used as the underlying theory of Cubical Agda [32]. There exists another cubical theory that develops a different structure of cubes, called *cartesian* cubical type theory, developed by Angiuli et al [4]. [6] This section follows syntax and notation in the style of Cubical Agda.

Path Types. Path types give the information for how two types are equal. Paths are maps out of an *interval* type I which has two elements i0 : I and i1 : I that are behaviorally equal (i.e. no function f : I → A can distinguish them) but are not definitionally equal. We say that functions f : I → A are *evidence* that f(i0) and f(i1) are equal in A. A path type specifies the behavior of their elements at i0 and i1:

$$\mathsf{PathP} : (A : I \to \mathsf{Type}\ l) \to A\ \mathsf{i0} \to A\ \mathsf{i1} \to \mathsf{Type}\ l$$

Notice that these types represent a *heterogeneous* equality, as the "endpoints" $a_0 : A\ \mathsf{i0}$ and $a_1 : A\ \mathsf{i1}$ have different types. Using the path types, *functional extensionality*, the property that pointwise equal functions are equal, is definable in the type theory. With a *homogeneous* instantiation of the path type, the statement and proof of funExt follows directly:

$$_ \equiv _ : \{A : \mathsf{Type}\ l\} \to A \to A \to \mathsf{Type}\ l$$
$$_ \equiv _\{A = A\}\ x\ y = \mathsf{PathP}(\lambda_ \to A)\ x\ y$$
$$\mathsf{funExt} : \{f\ g : A \to B\} \to ((x : A) \to f\ x \equiv g\ x) \to f \equiv g$$
$$\mathsf{funExt}\ p\ i\ x = p\ x\ i$$

Transport. *Transport* allows for coercions between two equal types. The type is stated as $\mathsf{transport} : A \equiv B \to A \to B$, and the term is instantiated with a transport primitive within the type theory.

Higher Inductive Types. *Higher Inductive Types* (HITs) are a generalization of inductive types and quotient types. Each constructor of the

[6] See Angiuli [1] or Angiuli et al. [2] for a detailed exposition of Cartesian Cubical Type Theory, and Cohen et al. [15] for a detailed exposition of De Morgan Cubical Type Theory.

type is not only a free generator but carries *paths between elements*. It can be used to take quotients of types by equivalence relations. One HIT that is especially useful is the *set quotient*, which quotients a type by an arbitrary relation, resulting in a set.

$$\begin{aligned}
&\mathsf{data}\ _/_\{A : \mathsf{Type}\} \to \{R : A \to A \to \mathsf{Type}\} \to \mathsf{Type}\ \mathsf{where} \\
&\qquad [_] : \{a : A\} \to A/R \\
&\qquad \mathsf{eq/} : \{a\ b : A\} \to \{r : R\ a\ b\} \to [a] \equiv [b] \\
&\qquad \mathsf{squash/} : \mathsf{isSet}(A/R).
\end{aligned}$$

underlying type, eq/, which equates all pairs of related elements, and squash/, which ensures that the resulting type is a set. Note that in the context of homotopy type theory, as with the univalence axiom, the computational behavior of HITs is not specified. In a cubical setting, however, the computational behavior can be described, as shown by Cavallo and Harper [13].

6.2 Isomorphisms are Not Enough

Univalence asserts that isomorphic types are equal, and thus Tabareau et al.'s *univalent parametricity* can transfer theorems between isomorphic types. This is a limitation because implementations that share the same abstract interface might not be isomorphic to each other. A standard queue interface can demonstrate this clearly. The naive implementation of a queue is the ListQueue, which enqueues an element to the head of the list, and dequeues an element from the tail of the list. A more efficient implementation is Okasaki's [26] BatchedQueue, which represents a queue with a tuple $\mathsf{Q} = \mathsf{ListA} \times \mathsf{ListA}$, where the first queue is used for enqueueing and the second is used to dequeue. This results in an amortized constant-time queue implementation, as opposed to the linear time complexity of the ListQueue implementation.

Observe that BatchQueue has the same extensional behavior as the ListQueue through the mapping appendReverse. appendReverse is a *structure-preserving* correspondence, as it commutes with enqueue and dequeue and preserves empty. Since representation independence states that two implementations sharing the same abstract interface are interchangeable when there is an operation-preserving correspondence, ListQueue and BatchedQueue are contextually equivalent. Yet these two implementations are not isomorphic to each other! The mapping, appendReverse is not injective, even for the simplest example: $([2], [])$ and $([], [2])$ both map to $[2]$. While univalence cannot be used for non-isomorphic instances, the *structure identity principle* (SIP),

```
ListQueue (A : Type) → Queue A
ListQueue A = queue (List A) [] _::_ last

BatchedQueue : (A : Type) → Queue A
BatchedQueue A =
  queue (List A x List A) ([], [])
       (fun x (xs, ys) → fastcheck (x :: xs, ys))
       (fun {(_, []) → nothing ; (xs, x :: ys) → just (
          fastcheck (xs, ys), x)})
  where
  fastcheck : {A : Type} → List A ⋆ List A → List A ⋆
      List A
  fastcheck (xs, ys) = if isEmpty ys then ([], reverse xs
      ) else (xs, ys)

appendReverse : {A : Type} → BatchedQueue A Q →
    ListQueue A Q
appendReverse (xs, ys) = xs ++ reverse ys
```

Figure 4: Contextually equivalent, but non-isomorphic implementations of Queue. appendReverse show the structure-preserving correspondence between the two implementations.

a result of univalence, can be used to alleviate this concern. Angiuli et al. [3] use the SIP in a cubical type theory setting (more specifically, Cubical Agda), to establish representation independence results. The strength of this work is two-fold: (1) this insight that transport between isomorphisms is not enough and using the SIP gives added expressivity for showing representation independence, and (2) using cubical type theory requires less engineering work, as there is no need to axiomatize univalence and declare tricky typeclass instances as in the Coq implementation of Tabareau et al. The key contribution results from the SIP paired with higher inductive types, along with a heterogeneous notion of equality using quasi-PERs results. Note that their use of the SIP is independent of their use of cubical type theory since SIP is a result of univalence.

In standard mathematical practice, properties over structures are assumed to be invariant up to isomorphisms on the same structure. The SIP brings this mathematical intuition to life, and the definition can vary depending on the notion of structure. Angiuli et al. commit to a definition of structure that uses dependent paths, which is convenient for the cubical setting. Structures are defined over a carrier type which has a notion of structure-preserving equivalence.

A *structure* is a function $S : \mathsf{Type} \to \mathsf{Type}$, and an *S-structure* is a dependent pair of a type and its application to the structure.

$$\mathsf{TypeWithStr\ S} = \Sigma[X \in \mathsf{Type}](S\ X)$$

An *S-structure-preserving* equivalence StrEquiv is a term with two S-structures and an equivalence between their underlying types, and it characterizes the proofs that the equivalence of underlying proofs is S-structure-preserving. $A \simeq [\ \iota\ ]B$ is the type of S-structure-preserving equivalences between A and B, shown as the following:

$$\mathsf{StrEquiv}\ S = (A\ B : \mathsf{TypeWithStr}\ S) \to \mathsf{fst}\ A \simeq \mathsf{fst}\ B \to \mathsf{Type}$$
$$A \simeq [\ \iota\ ]B = \Sigma[e \in \mathsf{fst}\ A \simeq \mathsf{fst}\ B](\iota\ A\ B\ e)$$

And finally, we say that (S, ι) defines a *univalent structure* if we have a term of the following type.

$$\begin{aligned}\mathsf{UnivalentStr}\ S\ \iota = \{A\ B : \mathsf{TypeWithStr}\ S\}&(e : \mathsf{fst}\ A \simeq \mathsf{fst}\ B)\\ &\to (\iota\ A\ B\ e) \simeq \mathsf{PathP}(\lambda i \to S(\mathsf{ua}\ e\ i))(\mathsf{snd}\ A)(\mathsf{snd}\ B)\end{aligned}$$

Escardo [18] also characterizes this same definition as the *standard notion of structure*, modified with the use of dependent paths. Given these definitions, the SIP can be defined, which can be trivially proven using univalence.

Theorem 4 (SIP)**.** *For S : Type $\to$ Type and ι : StrEquiv S, we have a term SIP : UnivalentStr $S\ \iota \to$ ($A\ B$: TypeWithStr S) $\to$ ($A \simeq [\ \iota\ ]\ B$) $\simeq$ ($A \equiv B$).*

This characterization of mathematical structure is similar to an algebraic theory [7], which is a pair of signatures and equations over structures $T = (\Sigma_T, \mathcal{E}_T)$. Similarly, structures are defined with a raw structure, which has operations on the carrier type, and propositional *axioms*.

As an example, consider a monad as a raw structure. One can pair this raw structure with an axiom stating that bind is unital and associative. Thus, a structure can be defined as this pair of raw structure and axiom and its structured equivalence will be the usual notion of a bijective monad homomorphism. This definition of the SIP and the use of set quotients defined by higher inductive types is sufficient for representation independence results.

Applying the SIP. The SIP can be used to transfer axioms between two non-equivalent implementations of Queues. Given a set A fixed, the

raw queue structure contains the empty queue, and the enqueue/dequeue functions.

$$\mathsf{RawQueueStructure}\ X = X * (A \to X \to X) * (X \to \mathsf{Maybe}(X * A))$$

Then, some axioms specifying the behavior of these operations, that Q is a set, and dequeue of the empty queue is nothing, and how dequeue followed by a enqueue operates. The axiom is as follows (the notation $\equiv$ is the notation for homogeneous paths as described in Section 6.1):

$$\begin{aligned}
&\mathsf{dequeueEnqueueAxiom}\ Q\ (\mathit{empty}, \mathit{enqueue}, \mathit{dequeue}) = \\
&\quad \forall a\ q \to \mathit{dequeue}\ (\mathit{enqueue}\ a\ q) \equiv \mathsf{just}\ (\mathsf{returnOrEnq}\ a\ (\mathit{dequeue}\ q)) \\
&\mathsf{returnOrEnq} : A \to \mathsf{Maybe}(Q \times A) \to Q \times A \\
&\mathsf{returnOrEnq}\ a\ \mathsf{nothing} = (\mathit{empty}, a) \\
&\mathsf{returnOrEnq}\ a\ (\mathsf{just}\ (q, b)) = (\mathit{enqueue}\ a\ q,\ b)
\end{aligned}$$

The RawQueueStructure and dequeueEnqueueAxiom can be combined to form a univalent QueueStructure. Recall the two contextually equivalent Queue implementations, ListQueue and BatchedQueue. These implementations do not form an isomorphism, and even worse, ListQueue satisfies dequeueEnqueueAxiom while BatchedQueue does not. For instance, $\mathit{dequeue}\ (\mathit{enqueue}\ c\ ([b, a], [])) \equiv \mathsf{just}(([], [b, c]), a)$, but $\mathsf{just}(\mathsf{returnOrEnq}\ c\ (\mathit{dequeue}\ ([b, a], []))) \equiv \mathsf{just}\ (([c], [b]), a)$.

Set quotients, with the use of the following HIT, can resolve this issue, by identifying any two BatchedQueues sent to the same list by appendReverse.

$$\begin{aligned}
&\mathsf{data\ BatchedQueueHIT : Type\ where} \\
&\quad Q\langle _, _ \rangle : \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{BatchedQueueHIT} \\
&\quad \mathsf{tilt} : \forall\ xs\ ys\ a \to Q\langle xs \mathbin{+\!\!+} [a],\ ys\rangle \equiv Q\langle xs,\ ys \mathbin{+\!\!+} [a]\rangle \\
&\quad \mathsf{squash} : \mathsf{isSet\ BatchedQueueHIT}
\end{aligned}$$

The BatchedQueueHIT is equipped with a QueueStructure. The *empty* queue is $Q\langle [], [] \rangle$, and *enqueue* and *dequeue* are defined to respect the tilt constructor. Since tilt can move elements between the end of the lists, this will ensure that any two BatchedQueueHIT with the same ListQueue can be identified. The structure-map between the structures, appendReverse, can be extended to an equivalence $\mathsf{BatchedQueueHIT} \simeq \mathsf{List}\ A$ which induces a raw queue structure on BatchedQueueHIT. Finally, applying the SIP transfers the ListQueue axioms to the quotiented BatchedQueue operations.

	Main Technique	Type theory	Result
Krishnaswami & Dreyer 2013	Internalized parametricity with *realizability semantics*	Extensional Calculus of Constructions	Adding **semantically well-typed terms** and free theorems
Tabareau et al. 2019	Univalent parametricity (Externalized parametricity with univalence)	Calculus of Inductive Constructions (CIC) with axiomatized univalence	**Automated** proof transport between isomorphic representations
Angiuli et al. 2021	Univalence (Structure Identity Principle)	De Morgan Cubical Type Theory	Proof transport between **non-isomorphic** representations

Figure 5: Summarized overview of the surveyed works

7 Conclusion

In this survey, we have overviewed several works that bring representation independence results in several dependent type theories. Dreyer and Krishnaswami's work on internalizing relational parametricity in the extensional Calculus of Constructions showed that induction principles and equality axioms can be soundly added back to the theory based on its realizability model. Tabareau et al.'s univalent parametricity offer transport à la carte, where basic univalent transport is strengthened with parametricity to allow for more proof transport and yield more efficient terms. Angiuli et al.'s use of the structure identity principle, a derivable property from univalence, allow for proof transport between non-isomorphic types which is useful for practical programming.

Regardless of how univalence is realized, Tabareau et al.'s univalent parametricity and Angiuli et al.'s SIP and set quotients strongly suggest that univalence plays an important role in bringing representation independence results. For a constructive account of type theory, Dreyer and Krishnaswami's QPER realizability model and the cubical type theory used by Angiuli et al. are salient. Although both Angiuli et al.'s work and Tabareau et al.'s work provide automated tools for proof transport in both of their frameworks, there is more burden in the tool for Tabareau et al.'s approach, because it engineers typeclass coercions to behave similarly to computation rules in cubical type theory. On the other hand, Angiuli et al.'s work is limited because of the *anticipation problem* where the abstract interface needs to be demonstrated a priori. (i.e. the representation independence result is in

terms of an abstract type), as opposed to Tabareau et al.'s approach which uses a heterogeneous relation to directly relate two concrete types with each other instead of relying on an abstract type. Concrete types enjoy definitional equality that abstract types do not (e.g. where an abstract + does not reduce on any input, but a concrete + defined on $\mathbf{N}$ will reduce on 0). In Angiuli et al.'s setting, the representation independence results considers programs that are parameterized by an interface that determines the notion of structure-preservation, which makes this concern less significant.

8 Acknowledgements

Thanks to Robert Harper, Stephanie Weirich, Yannick Zakowski, and Steve Zdancewic for their advice and comments.

References

[1] C. Angiuli. Computational semantics of cartesian cubical type theory. *To appear. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University*, 2019.

[2] C. Angiuli, G. Brunerie, T. Coquand, K.-B. Hou, R. Harper, and D. R. Licata. Syntax and models of cartesian cubical type theory.

[3] C. Angiuli, E. Cavallo, A. Mörtberg, and M. Zeuner. Internalizing Representation Independence with Univalence, 2020.

[4] C. Angiuli, R. Harper, et al. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[5] R. Atkey, N. Ghani, and P. Johann. A relationally parametric model of dependent type theory. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 503–515, 2014.

[6] H. P. Barendregt. Lambda calculi with types. 1992.

[7] A. Bauer. What is algebraic about algebraic effects and handlers? *arXiv preprint arXiv:1807.05923*, 2018.

[8] J.-P. Bernardy, T. Coquand, and G. Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 319:67–82, 2015.

[9] J.-P. Bernardy and M. Guilhem. Type-Theory in Color. *SIGPLAN Not.*, 48(9):61–72, Sept. 2013.

[10] J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and Dependent Types. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 345–356, New York, NY, USA, 2010. Association for Computing Machinery.

[11] J.-p. Bernardy, P. Jansson, and R. Paterson. Proofs for Free: Parametricity for Dependent Types. *J. Funct. Program.*, 22(2):107–152, Mar. 2012.

[12] J.-P. Bernardy and G. Moulin. A Computational Interpretation of Parametricity. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, page 135–144, USA, 2012. IEEE Computer Society.

[13] E. Cavallo and R. Harper. Higher inductive types in cubical computational type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–27, 2019.

[14] E. Cavallo and R. Harper. Parametric cubical type theory. *arXiv preprint arXiv:1901.00489*, 2019.

[15] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016.

[16] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs*, pages 147–162, Cham, 2013. Springer International Publishing.

[17] B. Dunphy and U. S. Reddy. Parametric Limits. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, page 242–251, USA, 2004. IEEE Computer Society.

[18] M. H. Escardó. Introduction to univalent foundations of mathematics with Agda. *arXiv preprint arXiv:1911.00580*, 2019.

[19] J.-Y. Girard. *Functional interpretation and 'e elimination of cuts in higher order arithmetic*. PhD thesis, 'E editor unknown, 1972.

[20] P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1-2):63–102, 2006.

[21] N. R. Krishnaswami and D. Dreyer. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In S. R. D. Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 432–451, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] G. Moulin. *Internalizing parametricity*. Department of Computer Science and Engineering, Chalmers University of . . . , 2016.

[23] G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. *ACM Sigplan Notices*, 44(9):135–148, 2009.

[24] nLab authors. axiom UIP. `http://ncatlab.org/nlab/show/axiom%20UIP`, Apr. 2021.

[25] A. Nuyts, A. Vezzosi, and D. Devriese. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.*, 1(ICFP), Aug. 2017.

[26] C. Okasaki. *Purely functional data structures*. Carnegie Mellon University, 1996.

[27] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, 1983.

[28] T. Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[29] N. Tabareau, É. Tanter, and M. Sozeau. The Marriage of Univalence and Parametricity. *arXiv preprint arXiv:1909.05027*, 2019.

[30] I. Takeuti. The Theory of Parametricity in Lambda Cube (Towards new interaction between category theory and proof theory). 1217:143–157, 2001.

[31] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[32] A. Vezzosi, A. Mörtberg, and A. Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31, 2021.

[33] V. Voevodsky. Univalent foundations of mathematics. In *International Workshop on Logic, Language, Information, and Computation*, pages 4–4. Springer, 2011.

[34] V. Voevodsky. The equivalence axiom and univalent models of type theory.(Talk at CMU on February 4, 2010). *arXiv preprint arXiv:1402.5556*, 2014.

[35] D. Vytiniotis and S. Weirich. Parametricity, type equality and higher-order polymorphism. 2010.

[36] P. Wadler. Theorems for Free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery.