Internal and Observational Parametricity for Cubical Agda

ANTOINE VAN MUYLDER, KU Leuven, Belgium
ANDREAS NUYTS, KU Leuven, Belgium
DOMINIQUE DEVRIESE, KU Leuven, Belgium

Two approaches exist to incorporate parametricity into proof assistants based on dependent type theory. On the one hand, parametricity translations conveniently compute parametricity statements and their proofs solely based on individual well-typed polymorphic programs. But they do not offer internal parametricity: formal proofs that any polymorphic program of a certain type satisfies its parametricity statement. On the other hand, internally parametric type theories augment plain type theory with additional primitives out of which internal parametricity can be derived. But those type theories lack mature proof assistant implementations and deriving parametricity in them involves low-level intractable proofs. In this paper, we contribute Agda --bridges: the first practical internally parametric proof assistant. We provide the first mechanized proofs of crucial theorems for internal parametricity, like the relativity theorem. We identify a high-level sufficient condition for proving internal parametricity which we call the structure relatedness principle (SRP) by analogy with the structure identity principle (SIP) of HoTT/UF. We state and prove a general parametricity theorem for types that satisfy the SRP. Our parametricity theorem lets us obtain one-liner proofs of standard internal free theorems. We observe that the SRP is harder to prove than the SIP and provide in Agda --bridges a shallowly embedded type theory to compose types that satisfy the SRP. This type theory is an observational type theory of logical relations and our parametricity theorem ought to be one of its inference rules.

CCS Concepts: • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: cubical type theory, parametricity, structure relatedness principle, Agda

ACM Reference Format:
Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. 2024. Internal and Observational Parametricity for Cubical Agda. *Proc. ACM Program. Lang.* 8, POPL, Article 8 (January 2024), 32 pages. https://doi.org/10.1145/3632850

1 INTRODUCTION

Theorems for free [Wadler 1989] are mathematical statements about polymorphic programs whose validity only depends on a program's type, not its implementation. Such theorems hold in programming languages that prevent polymorphic programs from inspecting their type arguments. This restriction forces polymorphic programs to behave *parametrically*, i.e., apply the same algorithm irrespective of the type they are invoked at.

For example, let us take a purely functional, polymorphic program taking two lists as input and outputting a single list, for an arbitrary type X (we use curly braces to indicate the presence of an implicit argument).

$$p : \forall\{X : \mathsf{Type}\} \to \mathsf{List}\, X \to \mathsf{List}\, X \to \mathsf{List}\, X \tag{1}$$

Authors' addresses: Antoine Van Muylder, KU Leuven, DistriNet, Belgium, antoine.vanmuylder@kuleuven.be; Andreas Nuyts, KU Leuven, DistriNet, Belgium, andreas.nuyts@kuleuven.be; Dominique Devriese, KU Leuven, DistriNet, Belgium, dominique.devriese@kuleuven.be.

ACM 2475-1421/2024/1-ART8
https://doi.org/10.1145/3632850

If p is parametric, its range of possible behaviors is considerably limited. Indeed, it cannot branch on concrete values of X like $X = \mathsf{Bool}$ and thus can only use its inputs xs, ys through the List interface: the resulting list $p\,xs\,ys$ must be obtained by interleaving, duplicating or omitting values from xs and ys. As a result, such a parametric program p should satisfy the following theorem:

$$\forall (A_0\,A_1 : \mathsf{Type})(xs\,ys : \mathsf{List}\,A_0)(f : A_0 \to A_1) \to \mathsf{map}\,f\,(p\,xs\,ys) \equiv p\,(\mathsf{map}\,f\,xs)\,(\mathsf{map}\,f\,ys) \quad (2)$$

The theorem holds when p is a list concatenation function, for instance. But in fact, the reasoning above applies for arbitrary parametric implementations of type $\forall\{X : \mathsf{Type}\} \to \mathsf{List}\,X \to \mathsf{List}\,X \to \mathsf{List}\,X$, which all satisfy the theorem. For this reason, the theorem is "free", i.e. obtained at zero cost.

Several approaches have been developed to study the theoretical aspects of free theorems and enable their use for program verification. The first formal definition of parametricity was given by [Reynolds 1983] for System F, a.k.a. the second-order polymorphic lambda calculus [Girard 1986, 1972; Reynolds 1974]. Reynolds defined parametricity for System F programs as a form of *preservation of relations*. More precisely, he proceeded in two steps. First, he defined a logical relation for System F types, that is, an inductively defined translation mapping any System F type τ into a set-theoretic relation $[\![\tau]\!]$ between inhabitants of the type. Second, he proved his *abstraction theorem* stating that every inhabitant $x : \tau$ is related to itself $x\,[\![\tau]\!]\,x$. For example[1], the abstraction theorem for $p : \forall\{X : \mathsf{Type}\} \to \mathsf{List}\,X \to \mathsf{List}\,X \to \mathsf{List}\,X$ implies:

$$\begin{aligned} &\forall A_0\,A_1\,(R : \mathsf{Rel}(A_0, A_1)) \to \forall\, xs_0\,xs_1 \text{ such that } \mathsf{ListRel}_R\,xs_0\,xs_1 \to \\ &\forall\, ys_0\,ys_1 \text{ such that } \mathsf{ListRel}_R\,ys_0\,ys_1 \to \mathsf{ListRel}_R\,(p\,xs_0\,ys_0)\,(p\,xs_1\,ys_1) \end{aligned} \quad (3)$$

The relation $\mathsf{ListRel}_R$ relates two lists iff they have the same length and their i-th elements are related by R for every i. Observe that Reynolds's relational definition of parametricity lets us successfully recover the free theorem (2) by setting $R(x, y)$ iff $f\,x \equiv y$ in property (3).

Reynolds's abstraction theorem is a metatheoretical statement about programs of System F. When verifying polymorphic programs in dependently typed proof assistants like Coq [The Coq development team 2022] or Agda [Agda Development Team 2023], such a metatheoretical statement is unsatisfactory. At best, it guarantees that the free theorem of interest can be added as an axiom in the proof assistant without jeopardizing the logical consistency of the system. The question then arises whether we can prove this free theorem *in* the proof assistant. To answer this question, two approaches have been developed in the literature: parametricity translations on one hand and dependent type theories with internal parametricity on the other.

Parametricity translations [Anand and Morrisett 2017; Bernardy et al. 2012; Keller and Lasson 2012] enhance Reynolds's logical relation $[\![-]\!]$ by acting both at the type level and at the term level. On the one hand, types of dependent type theory (DTT), say $T = (X : \mathsf{Type}) \to \mathsf{List}\,X \to \mathsf{List}\,X \to \mathsf{List}\,X$, are mapped to Reynolds's relational parametricity statement for that type, i.e., their logical relation $[\![T]\!]$. For T this is (basically) statement (3) but formulated inside DTT, contrary to what can be done for System F. On the other hand, terms of DTT are mapped to a *proof* of the logical relation at their type. For a program $p : T$ it is a proof of (3).

From the point of view of a proof assistant user, parametricity translations are convenient: solely based on an implementation $p : T$, they compute the statement of T's free theorem as well as its proof for p. Nonetheless, parametricity translations are not as powerful as they could be. Indeed

[1]For the sake of this example we imagine that System F features a List type.

they fail to provide proofs of free theorems like the following:

$$\begin{aligned}\text{glob-fthm} : \ & \forall(p : \{X : \mathsf{Type}\} \to \mathsf{List}\, X \to \mathsf{List}\, X \to \mathsf{List}\, X) \\ & \forall(A_0\, A_1 : \mathsf{Type})(xs\, ys : \mathsf{List}\, A_0)(f : A_0 \to A_1) \to \\ & \mathsf{map}\, f\, (p\, xs\, ys) \equiv p\, (\mathsf{map}\, f\, xs)\, (\mathsf{map}\, f\, ys)\end{aligned} \tag{4}$$

We call such a theorem *global* because the quantification on p is an object-level, or internal quantification (so not metatheoretical). Because of this internal quantification, theorems like glob-fthm are known to be logically independent from plain DTT [Booij et al. 2016; Boulier et al. 2017]. Another example of (an indirect consequence of) a global free theorem is the correctness of the Church encoding for the type of booleans $\mathsf{Bool} \simeq (X : \mathsf{Type}) \to X \to X \to X$. Since parametricity translations do not alter the logical expressivity of plain DTT, they cannot possibly validate global free theorems.

An alternative approach that does validate such global free theorems in DTT is the use of *internally parametric type theories* [Cavallo and Harper 2021; Moulin 2016; Nuyts and Devriese 2018; Nuyts et al. 2017]. In such type theories, free theorems can be proven from first principles, even theorems like (4). Those type theories draw inspiration from parametric denotational models of DTT (see e.g. [Atkey et al. 2014]) and augment plain DTT with so-called *parametricity primitives*: additional types, terms and equations (definitional equalities) from which free theorems can be derived. One such parametricity primitive, which appears in all internally parametric DTTs we know of, is the bridge type former (some systems use a different name). It axiomatizes the logical relation $[\![T]\!]$ at each type T. Internal parametricity then follows from the fact that programs preserve such bridges, similar to how parametric programs preserve relations.

Nonetheless, the cost of the extra logical expressivity granted by internally parametric DTTs is twofold. For one thing, only experimental, unpractical or incomplete implementations of such type theories exist (see e.g. [Cavallo 2020; Nuyts et al. 2017]). More fundamentally, even proving simple free theorems in those systems can be cumbersome. For example, Cavallo and Harper [2021] need about 25 lines of on-paper proof relying on their low-level parametricity primitives to establish that $\mathsf{Bool} \simeq (X : \mathsf{Type}) \to X \to X \to X$.

In this work we contribute a practical proof assistant and associated library combining the two approaches to parametricity in dependent type theory. Concretely, we first contribute the Agda --bridges proof assistant: an extension of the --cubical mode of the Agda proof assistant [Vezzosi et al. 2021]. It implements the parametricity primitives of Cavallo and Harper [2021] (CH), which we have adapted to the cubical type theory underlying Agda --cubical [Cohen et al. 2017]. In order to soundly emulate the substructurality of the bridge type former of CH and their variable-capturing equational theory, we build on the implementation of ticks [Mannaa and Møgelberg 2018] in Agda --guarded [Veltri and Vezzosi 2020, 2023] and let Agda --bridges raise freshness constraints on free variables at typechecking time. Furthermore, Agda --bridges reimplements the equational theory of the Kan operations hcomp, transp of Agda --cubical with respect to a novel, extended cofibration logic (a.k.a. face logic, see e.g. [Rose et al. 2022] for some background and references). Agda --bridges successfully compiles the full Agda --cubical standard library [Agda Community [n. d.]], offering evidence of its practicality and backwards compatibility. In particular, Agda --bridges conveniently enjoys strong extensionality principles like the univalence and function extensionality theorems provided by Agda --cubical. Within Agda --bridges, we give the first fully formal proofs of several theorems fundamental to internal parametricity, such as relativity [Cavallo and Harper 2021], the relational counterpart of univalence.

Agda --bridges can be used to prove (global) free theorems by hand using its low-level parametricity primitives, as is customary in internally parametric DTTs. However, experience showed

that such direct low-level proofs suffer from several drawbacks. First, familiarity with the CH parametricity primitives is required of the user. Second, these proofs lack compositionality: for example, it is unclear how to reuse a proof like (4) in order to shortcut a parametricity proof at a type built using $T = (X : \mathsf{Type}) \to \mathsf{List}\, X \to \mathsf{List}\, X \to \mathsf{List}\, X$ as a subterm. Third, these proofs are typically long and quickly get intractable as the complexity of the polymorphic target type grows.

To remedy this situation we first observe that the information required to carry out such proofs is exactly captured by a high-level metatheoretical principle which we call *the structure relatedness principle* (SRP). The SRP is a relational, dependent version of the *structure identity principle* (SIP; see e.g. Angiuli et al. [2021b] and Section 6.3). In essence, the SRP simply asserts that for each (dependent) type T, the bridge type of T is equivalent to the logical relation type at T. For a type T like (1), such an equivalence looks like the following, where p_0 and p_1 are polymorphic programs of type T and the left-hand side looks like (3):

$$\forall(A_0\, A_1 : \mathsf{Type})(R : A_0 \to A_1 \to \mathsf{Type}) \to \forall xs_0\, xs_1(xsr : \mathsf{ListRel}_R\, xs_0\, xs_1) \to \ldots$$
$$\simeq \quad \mathsf{Bridge}_{(X:\mathsf{Type})\to\mathsf{List}\,X\to\mathsf{List}\,X\to\mathsf{List}\,X}\ p_0\ p_1$$

Note that the SIP asks for similar characterizations, but for path types instead of bridge types.

The SRP at a certain (dependent) type, i.e., a characterization like the above, is enough to derive global free theorems for it. This is the content of our general parametricity theorem param (see Section 3.3.2), formulated in Agda --bridges for (dependent) types that satisfy the SRP. It is an internal abstraction theorem asserting that dependent functions preserve or act on logical relations. In practice, proofs of free theorems in Agda --bridges are one-liner invocations of param, if the appropriate type has been proven to satisfy the SRP.

Factoring the proof of a free theorem into an SRP proof obligation plus a call to the param theorem is already helpful. Such proofs are conceptually easier (with the same computational content) than their low-level counterparts and SRP proofs for simpler types compose to SRP proofs for more complex types. Nonetheless, proving the SRP for dependent types turns out to be challenging. Indeed we observe that some tools that facilitate SIP proofs do not translate to the relational setting: this includes the fundamental theorem of identity types (Theorem 3.5) and the useful fact that the proof-irrelevant fragment of a type can be ignored while proving the SIP for it (Theorem 3.6).

This motivates us to introduce a shallowly embedded domain-specific language (DSL) developed in Agda --bridges, letting the user combine types that validate the SRP. The mere act of writing a type in this DSL, amounts to the construction of an Agda --bridges type that satisfies the SRP, so that param can be used straightforwardly. Since it is a tool for composing dependent types, our DSL is in fact a dependent type theory itself, or rather a shallow embedding of a type theory. For the expert reader, our DSL consists of a (raw) category-with-families (CwF) structure on the category of types that satisfy the SRP. Our DSL can be seen as an observational type theory (OTT) [Altenkirch and Kaposi 2015; Altenkirch et al. 2022, 2007; Pujet and Tabareau 2022] of logical relations and our param theorem ought to be one of its inference rules. Indeed param can be seen as the relational analogue of the *ap* inference rule of OTT stating that open terms act on identifications (equalities). For this reason we call our DSL *relational observational type theory* (ROTT) and say that our param theorem is not just internal (to Agda --bridges) but also observational.

ROTT and param provide free theorems of practical and theoretical relevance as one-liners in Agda --bridges[2]: we prove theorem (4); we prove a scheme of Church encodings parametrized by a strictly positive functor; we give the first proof of Reynolds's abstraction theorem for (predicative) System F, using an internally parametric DTT (Agda --bridges) as a metatheory. This was not possible in [Nuyts et al. 2017]. We believe this is the first formal connection between an internally parametric system and Reynolds's relational parametricity. To summarize, our contributions are as follows:

• Agda --bridges: the first practical proof assistant with support for internal parametricity. It implements an adaptation of the internally parametric DTT of Cavallo and Harper [2021] (CH) to the type theory underlying Agda --cubical [Cohen et al. 2017] (CCHM). To faithfully implement the CH theory, Agda --bridges raises freshness constraints at typechecking time; the implementation of this feature builds on the implementation of ticks [Mannaa and Møgelberg 2018] in Agda --guarded [Veltri and Vezzosi 2020, 2023]. Agda --bridges reimplements the hcomp, transp operations of Agda --cubical with respect to an extended cofibration logic (a.k.a. face logic). It successfully compiles the full Agda --cubical standard library [Agda Community [n. d.]] and thus features the univalence and function extensionality theorems.
• Within Agda --bridges, we provide the first machine-checked proofs of several theorems of fundamental importance for internal parametricity, such as relativity.
• We identify a sufficient condition to obtain free theorems internally, the structure relatedness principle (SRP). We state and prove param: a general binary parametricity theorem (abstraction theorem) inside Agda --bridges, formulated for dependent types validating the SRP. Internal free theorems can be obtained out of param as one-liners.
• We identify objective reasons why, at a given type, proving the SRP is generally harder than proving the SIP. For these reasons, we provide ROTT: a shallowly embedded type theory for obtaining SRP proofs by merely writing a type, in the spirit of parametricity translations. Our param theorem ought to be one of the inference rules of ROTT. Internal parametricity proofs performed using ROTT and its param rule are user-friendly, compositional and concise.
• We demonstrate the use and generality of Agda --bridges, ROTT and param on practically and theoretically relevant examples.

Outline. This paper is structured as follows. In Section 2 we present the implementation and typing rules of Agda --bridges as well as its core theorems. More precisely: Section 2.1 is a brief reminder about cubical type theory and Agda --cubical; Section 2.2 discusses what makes Agda --bridges and the CH type theory substructural type systems. Sections 2.3, 2.4, 2.5 introduce the BridgeP, extent and Gel parametricity primitives; Section 2.6 and Section 2.7 prove core or basic theorems using Agda --bridges. The discussion about the Kan operations transp, mhcomp of Agda --bridges and their custom cofibration logic is rather postponed to Section 5 as they are not needed for the free theorems we present. In Section 3 we present a framework developed in Agda --bridges that provides abstractions to write user-friendly, modular and concise proofs of free theorems[2]. More precisely: In Section 3.1 we discuss the structure relatedness principle (SRP); In Section 3.2 we identify obstructions to the SRP; In Section 3.3 we present ROTT and its param rule. We use ROTT and apply param on various examples in Section 4. We conclude with related work in Section 6.

Conventions and notations. We use the term "logical relation" for (1) a relation R between structured types, compatible with the structure (e.g. a structure-preserving relation $R : M_0 \to M_1 \to \mathsf{Type}$ between two monoids); (2) a proof of relatedness for such a structured relation (e.g. a proof $\mathsf{pf} : R\, m_0\, m_1$ for some $m_0 : M_0$, $m_1 : M_1$); and (3) (more rarely) a relation defined by induction on the formation of types (like $[\![-]\!]$ above). Our choice to use the same term for (1) and (2) mirrors the situation for bridges. We assume that the reader is somewhat familiar with intensional dependent type theory and proof assistants based on it. The term "free theorem" designates consequences of relational parametricity for potentially non-graph relations R, at any given polymorphic type. For us, the equational theory of a type theory is the collection of its undirected definitional equalities and it includes β and η rules. We typically use Agda syntax to write types, programs and proofs. We ignore writing universe levels. We use curly braces for implicit arguments. Variable binding may

[2]All of our theorems are part of an Agda --bridges library https://github.com/antoinevanmuylder/bridgy-lib.

be denoted with $\lambda x.\, f$, $\lambda x \to f$ (Agda syntax) or sometimes just $x.f$. Logical relations between e.g. x_0 and x_1 are typically denoted with a postfix r notation xr. Bridges and paths between x_0 and x_1 are rather denoted with a double-letter notation xx. The ε symbol systematically ranges in $\{0, 1\}$.

2 THE INTERNAL PARAMETRICITY OF AGDA BRIDGES

Agda --bridges is a proof assistant extending the Agda --cubical proof assistant [Vezzosi et al. 2021]. Accordingly, the type theory that Agda --bridges implements, i.e., its primitives and their equational theory, is an extension of the type theory that Agda --cubical implements (called CCHM in the literature [Cohen et al. 2017]). A reminder about Agda --cubical appears in Section 2.1.

The type theory of Agda --bridges is an adaptation of the internally parametric DTT of Cavallo and Harper [2021] (referred to as the CH theory). In other words, Agda --bridges implements the primitives and equations specified by the CH theory (we mostly keep the same names), or rather relatively close variants. The CH theory is not entirely standard as it is a *substructural* (alternatively "affine") type theory. Indeed, most of its parametricity primitives have typing rules, including operational semantics, that can only be used if certain conditions on free variables are satisfied. This is discussed in Section 2.2. The first main difference between the Agda --bridges and CH theories lies in how they both handle substructurality. Our solution, *freshness typechecking constraints* on free variables, is discussed in Section 2.3. Note that the other main difference w.r.t. the CH theory is that both type theories extend different kinds of cubical type theories. The latter difference is rather discussed in Section 5.

In the CH theory, the bridge type former is postulated to represent *logical relations*: relations between types that respect their structure, or proofs of relatedness under such relations. Concrete examples of logical relations will appear below or can be consulted in [Hermida et al. 2013], for example. To ensure that bridges do uniquely correspond to logical relations, additional primitives called extent and Gel are postulated by the theory. Internal parametricity then refers to the fact that all programs preserve bridges, which are in one-to-one correspondence with logical relations. Accordingly, Agda --bridges features primitives BridgeP, extent and Gel whose implementation and rules are explained in Sections 2.3, 2.4, 2.5. Occurrences of these primitives in a program or type may generate freshness constraints at typechecking time.

In Section 2.6 and Section 2.7 we use the above primitives to derive within Agda --bridges core theorems for internal parametricity as well as the free theorem $\mathsf{Bool} \simeq (X : \mathsf{Type}) \to X \to X \to X$ in a low-level style.

2.1 The Cubical Fragment of Agda Bridges

Agda --cubical is an implementation of cubical type theory (CCHM) on top of the Agda proof assistant. Overall, cubical type theory modifies standard intensional type theory in several respects. First, it treats proofs of equality as if they were topological *paths* (see Section 2.1.1). Second, it features language primitives that let it realize *univalence* as a theorem (see Section 2.1.3). The latter property characterizes type equality as type equivalence (i.e. having a "bijection") and constitutes a defining aspect of *homotopy type theory* (HoTT) [Program 2013]. Earlier instances of HoTT assume univalence as an axiom instead. Third, these primitives include the so-called Kan operations called transport, hcomp in Agda --cubical. Our adaptation of the Kan operations is rather discussed in Section 5. Lastly, as an instance of HoTT, cubical type theory defines higher inductive data types (HITs) which we do not discuss in this work. We now remind the reader about paths, equivalences, univalence and other extensionality principles.

2.1.1 Paths. A major difference between Agda and Agda --cubical lies in how each system treats propositional equality. By contrast with the inductively defined equality types $\equiv$ of standard

intensional type theory, Agda --cubical defines equality in terms of a special postulated type called the *path interval* denoted $\mathbb{I}$ and equipped with two constants $\mathsf{i0}, \mathsf{i1}$ representing the interval's endpoints. A proof of equality of $a_0, a_1 : A$ is then by definition a function $aa : \mathbb{I} \to A$ such that $aa\ \mathsf{i0} = a_0$ and $aa\ \mathsf{i1} = a_1$ (definitionally). Proofs of equality are commonly called *paths* in the context of cubical type theory. In order to validate basic lemmas about path equality, $\mathbb{I}$ carries operations $\sim, \wedge, \vee$ for manipulating path variables. For instance, the map $\sim : \mathbb{I} \to \mathbb{I}$ inverts the two endpoints and can be used to prove symmetry of path equality: $\lambda\, aa\ i \to aa\ (\sim i) : a_0 \equiv a_1 \to a_1 \equiv a_0$.

To explain what are paths $p : a_0 \equiv_A a_1$ when A is constituted from dependent types (e.g. A is a Σ-type), an essentially unavoidable notion of *heterogeneous*, or *dependent* path emerges. For this reason Agda --cubical features types PathP of dependent paths. Given a line of types, that is, a function $A : I \to \mathsf{Type}$ and terms $a_0 : A\ \mathsf{i0}$ and $a_1 : A\ \mathsf{i1}$, one can form the type $\mathsf{PathP}_A\ a_0\ a_1 : \mathsf{Type}$ of heterogeneous or dependent paths between a_0 and a_1. While non-dependent paths $bb : b_0 \equiv_B b_1$ are functions $\mathbb{I} \to B$ with definitionally fixed endpoints b_0, b_1, *dependent* paths $aa : \mathsf{PathP}_A\ a_0\ a_1$ are *dependent* functions $(i : \mathbb{I}) \to A\ i$ with definitionally fixed endpoints a_0, a_1. In fact, the type $b_0 \equiv_B b_1$ of non-dependent paths in B is defined as the type $\mathsf{PathP}_{\lambda i \to B}\ b_0\ b_1$ of paths over a constant line $(\lambda i \to B : \mathbb{I} \to \mathsf{Type})$. Note that we will use the notation $i.\, A\ i := \lambda i \to A\ i$ to refer to lines of types in inlined code.

2.1.2 Equivalences. In Agda --cubical, an equivalence is a function $f : A_0 \to A_1$ satisfying the $\mathsf{isEquiv} : (A_0 \to A_1) \to \mathsf{Type}$ predicate. The type $A_0 \simeq A_1$ of equivalences between A_0 and A_1 is defined as $\Sigma[\, f \in (A_0 \to A_1)\,]\ \mathsf{isEquiv}\ f$. The exact definition of $\mathsf{isEquiv}$ can be consulted in [Program 2013] e.g., and we rather indicate here a sufficient condition for $\mathsf{isEquiv}\ f$ (our equivalences are always built this way). In order to prove $\mathsf{isEquiv}\ f$ it is sufficient to build an inverse for $f : A_0 \to A_1$, that is, a function $g : A_1 \to A_0$ such that $\forall\, a_0 \to g(f\ a_0) \equiv a_0$ and $\forall\, a_1 \to f(g\ a_1) \equiv a_1$. Note also that in order to prove that two equivalences $e_0, e_1 : A_0 \simeq A_1$ are equal, it suffices to prove that their underlying functions are equal.

2.1.3 Univalence and Other Extensionality Principles. It is commonplace in proof assistants to be faced with situations where one needs a more concrete representation of an equality type $a_0 \equiv_A a_1$ when the specific shape of A is known. For example, in order to prove that two (perhaps dependent) functions are equal it should be sufficient to prove that they are pointwise equal. In Agda --cubical, this principle admits a simple proof and is realized as an equivalence $\mathsf{funextEquiv} : ((a : A) \to f_0\ a \equiv_{B\ a} f_1\ a) \simeq (f_0 \equiv_{(a:A)\to B\ a} f_1)$.

Such characterizations of equality types $a_0 \equiv_A a_1$ at specific types A are called *extensionality principles*. Interestingly, Agda --cubical is expressive enough to guarantee the validity of similar extensionality principles for all primitive type formers, which we list here. The extensionality principle for Π (or $\to$ in the non-dependent case) is $\mathsf{funextEquiv}$. The principle for Σ (and $\times$) characterizes the path type $(p_0 \equiv_{\Sigma[\, a \in A\,]\, B\ a}\ p_1)$ as $\Sigma[\, aa \in (p_0\ .\mathsf{fst} \equiv_A p_1\ .\mathsf{fst})\,]\ \mathsf{PathP}_{i.\, B\ (aa\ i)}\ (p_0\ .\mathsf{snd})\ (p_1\ .\mathsf{snd})$. Note that $.\mathsf{fst}, .\mathsf{snd}$ extract values from (dependent) pairs. Similarly, extensionality principles for specific data and (even coinductive) record types can always be proven, and there also exists such a principle for the path type itself. Lastly, arguably the most important yet subtle primitive extensionality principle is *univalence*, which asserts that two types are equal if and only if they are equivalent, i.e., $\mathsf{univalence} : (A_0 \simeq A_1) \simeq (A_0 \equiv_{\mathsf{Type}} A_1)$. As all the other extensionality principles, univalence is obtained as a theorem in Agda --cubical.

2.2 Substructurality

As explained above, the CH theory is dubbed substructural (or alternatively "affine") because most of its parametricity primitives have typing rules, including operational semantics, that can only be used if certain conditions on free variables are satisfied. Reasons why these restrictions exist in

the first place appear later in this subsection. The CH theory manages to express such restrictions on free variables by using a *context restriction* operation $\Gamma \mapsto \Gamma \setminus x$ acting on contexts. Typically, restricted contexts $\Gamma \setminus x$ are strictly smaller than Γ and appear in the premises of certain typing rules of CH. Note that having premises featuring smaller contexts is not standard in dependent type theory. We now exemplify context restriction by looking at how the bridge type former is defined in the CH theory.

Substructurality in CH. Recall that the CH theory is an extension of cubical type theory. The first type former presented in CH is the bridge type former. The definition of the bridge type former is similar to that of the path type former as it is also based on the presence of an abstract interval BI called the *bridge interval*. The type BI is assumed to be equipped with two endpoints $\mathsf{bi0}, \mathsf{bi1} : \mathsf{BI}$ but no other operations (like $\sim, \wedge, \vee$ in the case of I, see Section 2.1). This implies that a term $\Gamma \vdash x : \mathsf{BI}$ in an arbitrary context is in fact always either a bridge variable $(x : \mathsf{BI}) \in \Gamma$ or an endpoint $\mathsf{bi0}, \mathsf{bi1}$.

Similar to the path case, the bridge introduction rule (at a closed type A, say) lets us build a bridge $\Gamma \vdash \lambda\, x.\, aa : \mathsf{Bridge}_A\, a_0\, a_1$ if a term $\Gamma, x : \mathsf{BI} \vdash aa : A$ is provided. However, the bridge and path type formers feature different elimination rules which we reproduce here:

$$\frac{\Gamma \vdash i : \mathsf{I} \qquad \Gamma \vdash aa : \mathsf{Path}_A\, a_0\, a_1}{\Gamma \vdash aa\, i : A} \qquad \frac{\Gamma \vdash x : \mathsf{BI} \qquad \Gamma \setminus x \vdash aa : \mathsf{Bridge}_A\, a_0\, a_1}{\Gamma \vdash aa\, x : A}\ \textsc{bdg-elim-ch}$$

As can be seen on the right, a bridge aa can be applied to a bridge interval term $\Gamma \vdash x : \mathsf{BI}$ only if aa typechecks in a different, *restricted* context $\Gamma \setminus x$. This restriction operation $\Gamma \mapsto \Gamma \setminus x$ is defined by structural induction on the context. Intuitively, if x is a variable (so not $\mathsf{bi0}, \mathsf{bi1}$) the context $\Gamma \setminus x$ is obtained from Γ by deleting the $x : \mathsf{BI}$ variable from Γ, as well as all those variables "strictly to the right" of x in Γ (whose type could legally contain x). This intuition is made formal in Section 2.3. In particular, the term $\lambda x.\, \mathsf{sq}\, x\, x$ that takes the diagonal of a square of bridges (i.e. a bridge between bridges), is ill-typed. The constraint appearing in BDG-ELIM-CH can be summarized by saying that bridges are only allowed to consume variables $x : \mathsf{BI}$ that are "fresh". Alternatively one can say that bridges, or the interval BI itself, are substructural, or affine.

The bridge elimination rule is one example of how context restriction is used to make BI substructural. In fact the majority of the rules appearing in the parametricity fragment of the CH theory feature restricted contexts. But knowing that context restriction can be used to enforce substructurality does not explain why the theory is substructural in the first place.

There are essentially two reasons for this, we refer to the CH article for more details. First, CH show that their substructural type theory admits a denotational model (in bicubical sets, see their article). This means that their type theory is logically consistent (relative to Set theory). In other words we can be sure that no logical contradiction can be derived using their primitives, an important requirement for a logic or a proof assistant. Second, several parametricity primitives of the theory (extent, Gel and the Kan operations) present a non standard *variable-capturing* equational theory that is well defined solely thanks to the substructurality of BI. For instance the β-rule of extent operates by identifying a free bridge variable $x : \mathsf{BI}$ in an appropriate argument m and by capturing the variable x in m, yielding an overall term with $\lambda x.\, m$ as a subterm. More information about capturing appears in Section 2.4.

Substructurality in Agda --bridges. Neither normal dependent Agda functions nor paths of Agda --cubical present an elimination rule with a restricted context. Accordingly, to minimize the implementation work and maximize the reuse of existing Agda infrastructure we do not use a context restriction operation: the premises of our rules do not have smaller contexts compared to their conclusion. For instance, eliminating a bridge aa at a bridge variable $\Gamma \vdash x : \mathsf{BI}$ only requires aa to typecheck in Γ (not $\Gamma \setminus x$). To remain sound w.r.t. the CH theory, Agda --bridges compensates by raising appropriate constraints on free variables when typechecking a rule featuring a premise

where $\Gamma \setminus x$ would appear in the CH theory. An example of such a rule is BDG-ELIM-CH. We call these constraints *freshness typechecking constraints*. Their definitions are given in Definition 2.1, 2.2. Such freshness constraints can be raised on different occasions during typechecking, which we list here.

- We are typechecking a bridge application $\Gamma \vdash aa\, x$.
- We are typechecking an affine function elimination $\Gamma \vdash f\, x$.
- We are computing (so reducing or comparing terms) and need capturing to occur.

Note that affine functions are exactly like bridges, but without fixed endpoints. Affine functions will be used to express the type of extent, for example. The first two cases are similar. If the raised constraint is found satisfiable, typechecking can continue and perhaps succeed. Else, a typechecking error occurs. In the third case, a freshness constraint called *semi-freshness* is raised. If it is found satisfiable computing goes on. Else, computing halts (no further reduction, or a failed comparison).

We now present the primitives of Agda --bridges: their typing rules and equational theory, details about their implementation as well as core theorems they guarantee. Following the above, several of these typing rules have premises that are (semi-) freshness constraints. Recall that all the theorems we obtain using Agda --bridges are available in our accompanying library.

2.3 Affine Functions and Bridges

First of all, Agda --bridges postulates the existence of a bridge interval type BI equipped with two endpoints bi0, bi1 : BI and no further operations. Next, we define the type former of affine functions. Bridges will essentially be affine functions with definitionally fixed endpoints.

2.3.1 Affine Functions. Affine functions are implemented as normal Agda dependent functions but their domain is BI and it carries what is called a tick annotation (building on [Veltri and Vezzosi 2020, 2023]). The type of non-dependent affine functions with codomain C is denoted $(@\mathsf{tick}\ x : \mathsf{BI}) \to C$ or $@\mathsf{BI} \to C$. We call an affine function $A : (@\mathsf{tick}\ x : \mathsf{BI}) \to \mathsf{Type}$ an (affine) line of types and such lines are sometimes denoted $x.\, A\, x$.

Given a line $A : (@\mathsf{tick}\ x : \mathsf{BI}) \to \mathsf{Type}$ one can form the type of dependent affine functions over A denoted $(@\mathsf{tick}\ x : \mathsf{BI}) \to A\, x$. Compared to normal dependent functions, the tick annotation has the net effect of raising an additional freshness constraint while typechecking the application of a function $f : (@\mathsf{tick}\ x : \mathsf{BI}) \to A\, x$ to a bridge variable $(x : \mathsf{BI})$ in a given context. That is to say, the following typing rule is implemented.

$$\frac{\Gamma \vdash f : (@\mathsf{tick}\ x : \mathsf{BI}) \to A\, x \qquad (x : \mathsf{BI}) \in \Gamma \qquad \mathsf{fresh}(f, x)}{\Gamma \vdash f\, x : A\, x}\ \textsc{tick-app}$$

The other rules of $(@\mathsf{tick}\ x : \mathsf{BI}) \to A\, x$ are those of normal dependent functions. Notice how this time the context in which f typechecks is not restricted. The constraint on free variables implied by the context restriction operation of (2.2) is instead expressed as a typechecking side condition denoted $\mathsf{fresh}(f, x)$ (understood to live at the same context Γ than f). We call the latter side condition a freshness constraint and it is defined as the following decidable condition on f, x.

Definition 2.1 (Freshness constraint). Setting $\Gamma = \Gamma_1, (x : \mathsf{BI}), \Gamma_2$, the freshness constraint $\mathsf{fresh}(f, x)$ is satisfied if for every free variable v of f one of the following holds:

- v appears in Γ_1 (i.e., $v \in \Gamma_1$)
- v appears in Γ_2 and is a path or a bridge variable[3], i.e., $(v : \mathsf{I}) \in \Gamma_2$ or $(v : \mathsf{BI}) \in \Gamma_2$.

We also adopt the convention that bi0, bi1 are always fresh for any term f.

[3]Additionally, v can be a variable witnessing the truth of a face constraint (Section 5.2) that does not mention x.

In more mundane terms, f and x satisfy the freshness constraint if f does not mention x as a free variable, nor any term variable (i.e. not a path/bridge variable) declared after x in the context Γ. In what follows the presence of a tick annotation will sometimes be abbreviated to an @ symbol, or omitted. To remain sound w.r.t. CH, all functions must use BI with this annotation.

We now explain the BridgeP, extent and Gel type formers. The typing rules implemented for those primitives are provided in Fig. 1. We make a few general remarks about these rules. The ε symbol systematically ranges over the indices $0, 1$. An equality judgment $\Gamma \vdash a = b : A$, or just $a = b$ when the context and types are clear, signifies that "the convertibility algorithm of the Agda --bridges typechecker concludes that a and b are definitionally equal". A reduction judgment $\Gamma \vdash a \mapsto b : A$, or just $a \mapsto b$ signifies that "the reduction algorithm of Agda --bridges finds that a reduces to b". More precisely, Agda uses weak head normalization and we denote the result of this algorithm on a by red a. To be more precise about our implementation, some of the rules of Fig. 1 contain occurrences of reduction red a. It is important to know that red is not a type-theoretic primitive but rather a hint that the corresponding argument should be reduced when firing the rule in question. In other words, occurrences of red in the rules must merely be seen as informative decorations providing more details about the implementation. The rules also present occurrences of variable captures $\langle x \rangle a$. Again, capturing is not an object-level program but rather a metatheoretical algorithm applying various substitutions and raising freshness constraints under the hood (see Section 2.4).

2.3.2 Bridges. As hinted above, the rules of BridgeP are essentially a replication of the rules of $(@x : \mathsf{BI}) \to A\, x$. According to the formation rule, and given an affine line of types $A : (@x : \mathsf{BI}) \to \mathsf{Type}$ as well as two endpoints $a_0 : A\, \mathsf{bi0}$ and $a_1 : A\, \mathsf{bi1}$ one can form the type $\mathsf{BridgeP}\, A\, a_0\, a_1 : \mathsf{Type}$ of heterogeneous or dependent bridges between a_0 and a_1 over the line A. We sometimes write the line A as an index as in $\mathsf{BridgeP}_A\, a_0\, a_1 : \mathsf{Type}$. Given a type $A : \mathsf{Type}$ and two inhabitants a_0, a_1, the type of non dependent bridges between a_0, a_1 is defined as $\mathsf{Bridge}_A\, a_0\, a_1 = \mathsf{BridgeP}_{x.A}\, a_0\, a_1$. Furthermore, eliminating a bridge at a bridge variable can only be done if a freshness constraint is satisfied, similar to the TICK-APP rule of Section 2.3.1. Lastly, when typechecking a bridge $\Gamma \vdash aa : \mathsf{BridgeP}_A\, a_0\, a_1$, the typechecker will verify that $aa\, \mathsf{bi}\varepsilon$ and a_ε are definitionally equal. Conversely, the expression $aa\, \mathsf{bi}\varepsilon$ will reduce to a_ε for a typechecked bridge.

So far only one example of a bridge can be built in an empty context. If A is a closed type inhabited by a, one can build the reflexivity bridge at A as $\lambda x.\, a : \mathsf{Bridge}_A\, a\, a$. To build examples of bridges different from reflexivity at function types $(a : A) \to B\, a$, the extent primitive is introduced.

2.4 The Extent Primitive

We now turn our attention to the rules and implementation of the extent parametricity primitive. The rules of extent are displayed in Fig. 1. The sole purpose of extent is to guarantee the validity of a certain *relational* extensionality principle: a principle akin to funextEquiv but characterizing bridges between functions rather than paths. We call this principle extentEquiv and it reads as:

$$((a_0\, a_1 : A)(aa : \mathsf{Bridge}_A\, a_0\, a_1) \to \mathsf{BridgeP}_{x.B(aa\,x)}(f_0\, a_0)(f_1\, a_1)) \;\simeq\; \mathsf{Bridge}_{(a:A)\to B\,a}\, f_0\, f_1$$

This equivalence asserts that two functions f_0, f_1 are related by a bridge if and only if they are pointwise related, i.e., they map related inputs to related outputs, where "related" signifies the existence of an appropriate bridge. In other words it makes the bridge type former behave as a type former of logical relations, at $(a : A) \to B\, a$. Note that there exists a slight generalization of this principle characterizing the type of heterogeneous bridges $\mathsf{BridgeP}_{x.\,(a:A\,x)\to B\,x\,a}\, f_0\, f_1$ instead, which we still call extentEquiv.

$$\frac{\Gamma \vdash A : @\mathsf{BI} \to \mathsf{Type} \qquad \Gamma \vdash a_\varepsilon : A\,\mathsf{bi}\varepsilon}{\Gamma \vdash \mathsf{BridgeP}_A\ a_0\ a_1 : \mathsf{Type}}\ \textsc{bdg-form} \qquad \frac{\Gamma \vdash aa : (@x : \mathsf{BI}) \to A\,x \qquad \Gamma \vdash aa\,\mathsf{bi}\varepsilon = a_\varepsilon}{\Gamma \vdash \lambda x.\, aa\,x : \mathsf{BridgeP}_A\ a_0\ a_1}\ \textsc{bdg-intro}$$

$$\frac{\Gamma \vdash aa : \mathsf{BridgeP}_A\ a_0\ a_1 \qquad \Gamma \vdash x : \mathsf{BI} \qquad \mathsf{fresh}(aa, x)}{\Gamma \vdash aa\,x : A\,x}\ \textsc{bdg-elim} \qquad \frac{\Gamma \vdash aa : \mathsf{BridgeP}_A\ a_0\ a_1}{\Gamma \vdash aa\,\mathsf{bi}\varepsilon \mapsto a_\varepsilon}\ \textsc{bdg-}\partial$$

$$\frac{\Gamma \vdash aa_\varepsilon : \mathsf{BridgeP}_A\ a_0\ a_1 \quad \Gamma, @x : \mathsf{BI} \vdash aa_0\,x = aa_1\,x : A\,x}{\Gamma \vdash aa_0 = aa_1 : \mathsf{BridgeP}_A\ a_0\ a_1}\ \textsc{bdg-}\eta \qquad \frac{\Gamma \vdash aa : (@x : \mathsf{BI}) \to A\,x \qquad \Gamma \vdash y : \mathsf{BI} \qquad \mathsf{fresh}(aa, y)}{\Gamma \vdash (\lambda x.\, aa\,x)\ y \mapsto aa\ y : A\ y}\ \textsc{bdg-}\beta$$

$$\frac{\Gamma \vdash A : @\mathsf{BI} \to \mathsf{Type} \qquad (x : \mathsf{BI}) \in \Gamma \qquad \Gamma \vdash a : A\,x \qquad \mathsf{sfresh}(a, x)}{\Gamma \vdash \langle x \rangle a : (@y : \mathsf{BI}) \to A\ y \quad \mathsf{fresh}(\langle x \rangle a, x) \quad \Gamma \vdash (\langle x \rangle a)\ x = a : A\,x}\ \textsc{cap}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : @\mathsf{BI} \to \mathsf{Type} \qquad \Gamma \vdash B : (@x : \mathsf{BI})(a : A\,x) \to \mathsf{Type} \qquad \Gamma \vdash f_\varepsilon : (a_\varepsilon : A\,\mathsf{bi}\varepsilon) \to B\,\mathsf{bi}\varepsilon\,a_\varepsilon \\ \Gamma \vdash fr : (a_0 : A\,\mathsf{bi0})(a_1 : A\,\mathsf{bi1})(aa : \mathsf{BridgeP}_A\ a_0\ a_1) \to \mathsf{BridgeP}_{x.\,B\,x\,(aa\,x)}\ (f_0\ a_0)\ (f_1\ a_1)\end{array}}{\Gamma \vdash \mathsf{extent}\ \{A\}\ \{B\}\ f_0\ f_1\ fr : (@x : \mathsf{BI})(a : A\,x) \to B\,x\,a}\ \textsc{ext}$$

$$\frac{\text{premises of } \textsc{ext} \qquad \Gamma \vdash a_\varepsilon : A\,\mathsf{bi}\varepsilon}{\Gamma \vdash \mathsf{extent}\ f_0\ f_1\ fr\,\mathsf{bi}\varepsilon\ a_\varepsilon \mapsto f_\varepsilon\ a_\varepsilon}\ \textsc{ext-}\partial$$

$$\frac{\text{premises of } \textsc{ext} \qquad (x : \mathsf{BI}) \in \Gamma \qquad \Gamma \vdash a : A\,x \qquad \mathsf{sfresh}(\mathsf{red}\,a, x)}{\Gamma \vdash \mathsf{extent}\ f_0\ f_1\ fr\,x\,a \mapsto fr\,(\langle x \rangle a\ \mathsf{bi0})(\langle x \rangle a\ \mathsf{bi1})(\langle x \rangle(\mathsf{red}\,a))\ x : B\,x\,a}\ \textsc{ext-}\beta$$

$$\frac{\Gamma \vdash A_\varepsilon : \mathsf{Type} \qquad \Gamma \vdash R : A_0 \to A_1 \to \mathsf{Type}}{\Gamma \vdash \mathsf{Gel}\,A_0\,A_1\,R : (@x : \mathsf{BI}) \to \mathsf{Type}}\ \textsc{Gel-form} \qquad \frac{}{\Gamma \vdash \mathsf{Gel}\,A_0\,A_1\,R\,\mathsf{bi}\varepsilon \mapsto A_\varepsilon : \mathsf{Type}}\ \textsc{Gel-}\partial$$

$$\frac{\Gamma \vdash a_\varepsilon : A_\varepsilon \qquad \Gamma \vdash ar : R\,a_0\,a_1}{\Gamma \vdash \mathsf{gel}\ \{A_0\}\ \{A_1\}\ \{R\}\ a_0\ a_1\ ar : (@x : \mathsf{BI}) \to \mathsf{Gel}\,A_0\,A_1\,R\,x}\ \textsc{Gel-intro} \qquad \frac{}{\Gamma \vdash \mathsf{gel}\ a_0\ a_1\ ar\ \mathsf{bi}\varepsilon \mapsto a_\varepsilon : A_\varepsilon}\ \textsc{gel-}\partial$$

$$\frac{\Gamma \vdash Q : (@x : \mathsf{BI}) \to \mathsf{Gel}\,A_0\,A_1\,R\,x}{\Gamma \vdash \mathsf{ungel}\ \{A_0\}\ \{A_1\}\ \{R\}\ Q : R\ (Q\ \mathsf{bi0})\ (Q\ \mathsf{bi1})}\ \textsc{Gel-elim} \qquad \frac{\Gamma \vdash a_\varepsilon : A_\varepsilon \qquad \Gamma \vdash ar : R\,a_0\,a_1}{\Gamma \vdash \mathsf{ungel}\ (\lambda x.\, \mathsf{gel}\ a_0\ a_1\ ar\ x) \mapsto ar : R\,a_0\,a_1}\ \textsc{Gel-}\beta$$

$$\frac{\begin{array}{c}\Gamma \vdash g_\varepsilon : \mathsf{Gel}\,A_0\,A_1\,R\,x \qquad (x : \mathsf{BI}) \in \Gamma \qquad \mathsf{sfresh}(\mathsf{red}\,g_\varepsilon, x) \\ \Gamma \vdash \langle x \rangle g_0\ \mathsf{bi}\varepsilon = \langle x \rangle g_1\ \mathsf{bi}\varepsilon \qquad \Gamma \vdash \mathsf{ungel}\ (\langle x \rangle(\mathsf{red}\,g_0)) = \mathsf{ungel}\ (\langle x \rangle(\mathsf{red}\,g_1)) : R\ (\langle x \rangle g_0\ \mathsf{bi0})\ (\langle x \rangle g_0\ \mathsf{bi1})\end{array}}{\Gamma \vdash g_0 = g_1 : \mathsf{Gel}\,A_0\,A_1\,R\,x}\ \textsc{Gel-}\eta$$

Fig. 1. Rules of the BridgeP, extent and Gel primitives of Agda --bridges. Some premises are omitted.

In order to validate the left-to-right direction of the above principle, Agda --bridges postulates the existence of the extent primitive (see also the EXT rule of Fig. 1). Note that the type of extent rather ends with a type of affine functions. This difference is essentially cosmetic thanks the the EXT-∂ rule of extent.

```
extent : ∀ {A : (@tick x : BI) → Type} {B : (@tick x : BI) (a : A x) → Type}
  (f₀ : (a₀ : A bi0) → B bi0 a₀) (f₁ : (a₁ : A bi1) → B bi1 a₁)
  (fr : (a₀ : A bi0) (a₁ : A bi1) (aa : BridgeP A a₀ a₁) → BridgeP (λ x → B x (aa x)) (f₀ a₀) (f₁ a₁))
  (@tick x : BI) (a : A x) → B x a
```

It remains to upgrade this map into an actual equivalence extentEquiv. Validating the right-to-left direction of extentEquiv is possible without assuming further primitives. Indeed, assuming a bridge $ff : \mathsf{Bridge}_{(a:A)\to B\,a}\ f_0\ f_1$, we can easily build a function of the appropriate type $\lambda a_0\ a_1\ aa \to \lambda x.\ ff\ x\ (aa\,x)$. Furthermore, one of the inverse conditions can be obtained using a "propositional η-rule" theorem proved for extent. The other inverse condition relies on the β-rule of extent (see EXT-β in Fig. 1). The exact proof can be consulted in our accompanying library, or in the CH paper.

The β-rule of extent is not a standard type theoretic rule, as it works by *capturing* (see CAP rule) the bridge variable argument $x : \mathsf{BI}$ of extent in its principal argument $a : A\,x$. Capturing can only be performed if a specific freshness constraint (semi-freshness, denoted $\mathsf{sfresh}(a, x)$) is satisfied and thus EXT-β only fires in that case. We now explain what semi-freshness is and how capturing is implemented. Note that several other parametricity primitives of Agda --bridges like Gel and the Kan operations make use of capturing in their equations.

Definition 2.2 (Semi-freshness constraint). Given a context $\Gamma = \Gamma_1, (x : \mathsf{BI}), \Gamma_2$ and a term $\Gamma \vdash a$, the constraint $\mathsf{sfresh}(a, x)$ is satisfied if for every free variable v of a one of the following holds:

- $v \in \Gamma_1$ **or** $v = x$,
- $(v : \mathsf{I}) \in \Gamma_2$ or[3] $(v : \mathsf{BI}) \in \Gamma_2$. We define Υ_2 as the variables v satisfying this clause.

If $\Gamma_1, (x : \mathsf{BI}), \Gamma_2 \vdash a : A\,x$ and $\mathsf{sfresh}(a, x)$ then a is in fact a weakening $a = a'[\pi]$ of a term $\Gamma_1, (x : \mathsf{BI}), \Upsilon_2 \vdash a' : A'\,x$ along $\pi : \Gamma_1, (x : \mathsf{BI}), \Gamma_2 \to \Gamma_1, (x : \mathsf{BI}), \Upsilon_2$[4]. Moreover, there is a well-typed substitution $\rho : \Gamma_1, (x : \mathsf{BI}), \Gamma_2, (y : \mathsf{BI}) \to \Gamma_1, (x : \mathsf{BI}), \Upsilon_2$ defined by $\rho = (\mathsf{id}_{\Gamma_1}, y/x, \mathsf{id}_{\Upsilon_2})$, so that we have $\Gamma_1, (x : \mathsf{BI}), \Gamma_2, (y : \mathsf{BI}) \vdash a'[\rho] : A'[\rho]\,y$. Agda --bridges will shortcut this by constructing a potentially ill-typed substitution $\sigma = (\mathsf{id}_{\Gamma_1}, y/x, \mathsf{id}_{\Gamma_2}) : \Gamma_1, (x : \mathsf{BI}), \Gamma_2, (y : \mathsf{BI}) \to \Gamma_1, (x : \mathsf{BI}), \Gamma_2$ which has the property that $\pi \circ \sigma = \rho$, and therefore $\Gamma_1, (x : \mathsf{BI}), \Gamma_2, (y : \mathsf{BI}) \vdash a[\sigma] = a'[\rho] : A[\sigma]\,y$ is well-typed. By freshness w.r.t. x, $A[\sigma] = A$ and by lambda abstraction we obtain $\Gamma \vdash \lambda y.\, a[\sigma] : (@y : \mathsf{BI}) \to A\,y$. The latter is the definition of capturing $\Gamma \vdash \langle x \rangle a : (@y : \mathsf{BI}) \to A\,y$.

2.5 Gel Types

The univalence theorem stated in Section 2.1.3 posits that paths at the universe $AA : A_0 \equiv A_1$ uniquely correspond to type equivalences $A_0 \simeq A_1$. Analogously, the *relativity* theorem asserts that *bridges* at the universe uniquely correspond to *relations*.

$$\mathsf{relativity} : \quad (A_0 \to A_1 \to \mathsf{Type}) \;\simeq\; \mathsf{Bridge}_{\mathsf{Type}}\, A_0\, A_1$$

Similar to extent, Agda --bridges postulates the existence of Gel in order to validate the left-to-right direction of relativity. Thus Gel has the following type (see also GEL-FORM in Fig. 1).

$$\mathsf{Gel} : \forall\, (A_0\; A_1 : \mathsf{Type})\, (R : A_0 \to A_1 \to \mathsf{Type})\, (@\mathsf{tick}\; x : \mathsf{BI}) \to \mathsf{Type}$$

The type of Gel merely ends with a type of affine functions $(@x : \mathsf{BI}) \to \mathsf{Type}$ which can be turned into a bridge type thanks to the GEL-∂ rule.

To upgrade this map into the relativity equivalence, we first need an inverse candidate. From a bridge between two types $AA : \mathsf{Bridge}_{\mathsf{Type}}\, A_0\, A_1$ we obtain the following relation $\lambda\, a_0\; a_1 \to \mathsf{BridgeP}_{x.\,AA\,x}\; a_0\; a_1 : A_0 \to A_1 \to \mathsf{Type}$. This relation between the types A_0 and A_1 holds for a_0, a_1 exactly when there is a dependent bridge over AA between them. We also need to provide two inverse conditions.

2.5.1 The First Inverse Condition. Regarding the first condition, using function extensionality one has to show $\forall (R : A_0 \to A_1 \to \mathsf{Type})(a_0 : A_0)(a_1 : A_1) \to \mathsf{BridgeP}_{x.\,\mathsf{Gel}\,A_0\,A_1\,Rx}\; a_0\; a_1 \equiv R\, a_0\; a_1$. By univalence, it can be reduced to proving an equivalence $\mathsf{BridgeP}_{x.\,\mathsf{Gel}\,A_0\,A_1\,R\,x}\; a_0\; a_1 \simeq R\, a_0\; a_1$. One could call this equivalence a (dependent) relational extensionality principle for Gel. Constructing both directions of this particular equivalence is precisely the role played by the introduction and elimination primitives of Gel, called gel and ungel, respectively (see GEL-INTRO, GEL-ELIM). The fact that gel and ungel are mutual inverses is in turn guaranteed by the equations governing those primitives, called the η-rule and the β-rule of Gel (see GEL-β, GEL-η). Note that the η-rule of Gel

[4]Because $A\,x$ is a well-typed bridge application, A is fresh w.r.t. x and therefore $A = A'[\pi]$.

makes use of capturing, as was the case for EXT-β. This means in particular that a semi-freshness constraint is checked when comparing two inhabitants of Gel.

2.5.2 The Second Inverse Condition. Regarding the second inverse condition needed to prove relativity, one has to show $\forall(AA : \mathsf{Bridge}\, A_0\, A_1) \to \mathsf{Gel}\, A_0\, A_1\, (\lambda a_0 a_1.\, \mathsf{BridgeP}_{x.\, AA\, x} a_0 a_1) \equiv AA$. Formally proving this lemma in Agda --bridges was far from trivial (see our accompanying library). In their paper and technical report, Cavallo and Harper [2019, 2021] sketch a proof of this lemma based on a relational extensionality principle for *equivalences* whose lengthy proof they also sketch. The principle is a characterization of the type of heterogeneous bridges between equivalences and its formulation is somewhat comparable to extentEquiv. We merely indicate here that our formal proof involves the pasting of several 2-dimensional paths in Type, whose construction relies on the path interval operations ~, ∧, ∨ provided by Agda --cubical.

2.6 Other Relational Extensionality Principles

The primitives extent and Gel, gel, ungel we have implemented grant relational extensionality principles for the Π type former and for the universe Type, respectively. It turns out that their addition alone ensures the validity of similar principles for the other primitive type formers of the theory. For instance, as is the case for paths, a bridge at $\Sigma[\, a \in A\,]\, B\, a$ between pairs $(a_0, b_0), (a_1, b_1)$ can equivalently be regarded as a bridge in the base $aa : \mathsf{Bridge}_A\, a_0\, a_1$ together with a heterogeneous bridge over it $bb : \mathsf{BridgeP}_{x.\, B(aa\, x)}\, b_0\, b_1$. There also exist relational extensionality principles for the path type ≡ and the Bridge type itself. Essentially, those two principles reflect the fact that is is always possible to swap the order of bridge and path variables in the context.

Relational extensionality principles for specific inductive data types can also be stated and proved in Agda --bridges. For instance, in their paper CH prove such a principle for the type of booleans Bool. The principle expresses that Bool is *bridge-discrete*, that is, the only bridges in Bool are the ones corresponding to its paths: $b_0 \equiv_{\mathsf{Bool}} b_1 \simeq (\mathsf{Bridge}_{\mathsf{Bool}}\, b_0\, b_0)$. Since Bool has no non-reflexivity paths (i.e., is an h-set in HoTT parlance), there are only two bridges in Bool: the reflexivity bridges at true and false. We have adapted the argument of CH to prove in Agda --bridges a similar principle for the List parametrized data type. More precisely, it is a *dependent* extensionality principle as it characterizes the type of *heterogeneous* bridges $\mathsf{BridgeP}_{x.\mathsf{List}(AA\, x)}\, as_0\, as_1$ between two lists $(as_0 : \mathsf{List}\, A_0), (as_1 : \mathsf{List}\, A_1)$ where $AA : \mathsf{Bridge}\, A_0\, A_1$.

```
ListRel : ∀ {A₀ A₁ : Type} (R : A₀ → A₁ → Type) → List A₀ → List A₁ → Type
ListRel R [] [] = Unit
ListRel R [] (_ :: _) = ⊥
ListRel R (_ :: _) [] = ⊥
ListRel R (a₀ :: as₀) (a₁ :: as₁) = R a₀ a₁ × ListRel R as₀ as₁

ListvsBridgeP : ∀ {A₀ A₁ : Type} (AA : Bridge A₀ A₁) (as₀ : List A₀) (as₁ : List A₁) →
  ListRel (BridgeP (λ x → AA x)) as₀ as₁ ≃ BridgeP (λ x → List (AA x)) as₀ as₁
```

The principle essentially expresses that a bridge between lists is a list of bridges. Indeed, ListRel R is an inductively defined relation between the types List A_0 and List A_1 which holds for as_0, as_1 exactly when the latter lists have the same size and exhibit R-related values at each index.

2.7 Low-Level Parametricity

We are now ready to prove theorems for free from first principles in Agda --bridges. It is customary in type theories with internal parametricity (incl. CH) to prove free theorems by directly appealing

```
lowChurchBool : (∀ (X : Type) → X → X → X) ≃ Bool
lowChurchBool = isoToEquiv (iso chToBool boolToCh (λ { true → refl ; false → refl })
  λ k → funExt λ A → funExt λ t → funExt λ f → param-prf k A t f)
  where
    boolToCh : Bool → (∀ (X : Type) → X → X → X)
    boolToCh true X xt xf = xt
    boolToCh false X xt xf = xf

    chToBool : (∀ (X : Type) → X → X → X) → Bool
    chToBool k = k Bool true false

    module CH-inverse-cond (k : ∀ (X : Type) → X → X → X) (A : Type) (t f : A) where
      R : Bool → A → Type
      R = λ b a → (boolToCh b A t f) ≡ a
      k-Gelx : (@tick x : BI) → Gel Bool A R x → Gel Bool A R x → Gel Bool A R x
      k-Gelx x = k (Gel Bool A R x)
      k-Gelx-gel-gel : (@tick x : BI) → Gel Bool A R x
      k-Gelx-gel-gel x = k-Gelx x (gel true t (refl) x) ((gel false f (refl) x))
      asBdg : BridgeP (λ x → Gel Bool A R x) (k Bool true false) (k A t f)
      asBdg x = k-Gelx-gel-gel x
      param-prf : R (k Bool true false) (k A t f)
      param-prf = ungel {R = R} λ x → asBdg x
    open CH-inverse-cond
```

Fig. 2. A low-level proof of a free theorem.

to the available low-level parametricity primitives. This is unsurprising since, after all, those primitives have been added precisely for that purpose.

Such a low-level proof of a free theorem in Agda --bridges appears in Fig. 2. The lowChurchBool theorem asserts that Bool admits a Church encoding, i.e., that this equivalence holds: $(X : \mathsf{Type}) \to X \to X \to X \simeq \mathsf{Bool}$. It is a faithful reproduction of the proof appearing in [Cavallo and Harper 2021]. We provide a high-level description of the proof and refer to the latter for more detailed explanations. To build an equivalence, it is sufficient to provide two maps and two inverse conditions. This is the content of the isoToEquiv lemma. The two candidate inverses are defined in a where block below and are called boolToCh and chToBool. The first inverse condition can simply be proven by induction. Using function extensionality, the second inverse condition asks that for every $k : (X : \mathsf{Type}) \to X \to X \to X$, this equality holds $\mathsf{boolToCh}(\mathsf{chToBool}\, k)\, A\, t\, f \equiv k\, A\, t\, f$. Note that the universal quantification on k appears inside the system (with an Agda Π-type). The logic of Agda --cubical alone would not be sufficient to warrant this result (see Section 1). Therefore the internal parametricity of Agda --bridges must be used. All calls to parametricity primitives are isolated in a separate module called CH-inverse-cond. The last lemma of this module param-prf implies the second inverse condition.

We observe that such low-level proofs suffer from several defects. First, the user of Agda --bridges wanting to reproduce this style of proofs must have a good familiarity with the parametricity primitives provided by Agda --bridges, including their inner workings. But these primitives use advanced or non-standard type-theoretic notions like freshness and capturing. This makes for hard and non user-friendly proofs. Second these proofs lack compositionality. Indeed, we have proved in Fig. 2 a free theorem param-prf about polymorphic programs of type $T = (X : \mathsf{Type}) \to X \to X \to X$. We expect other free theorems to hold at this type and it is unclear at first glance if param-prf could be reused to achieve that. In fact we even would like to be able to reuse the

latter parametricity proof to shortcut proofs of free theorems at types having $- \to - \to -$ as a subexpression. Lastly, these proofs are typically long even for seemingly simple examples of free theorems. Furthermore their complexity quickly gets intractable when the size of the target type T grows (there are several reasons for this, discussed in the next section).

All in all, these drawbacks motivated us to develop in Agda --bridges a library providing user-friendly, compositional and short proofs of free theorems. This is the content of Section 3.

3 THE OBSERVATIONAL PARAMETRICITY OF AGDA BRIDGES

As explained in Section 2.7, low-level proofs of internal free theorems are unsatisfactory in several respects. We improve these low-level proofs in two steps.

Our first improvement stems from the observation that, in order to make use of internal parametricity, it is always sufficient to prove appropriate relational extensionality principles. More precisely, we argue that obtaining internal free theorems for an Agda --bridges program $p : (\gamma : \Gamma) \to T\ \gamma$ can be reduced to providing (dependent) relational extensionality principles for p's domain and codomain, so characterizations of their (dependent) bridge types as types of actual logical relations: an equivalence $\eta^{\Gamma} : \ldots \simeq \mathsf{Bridge}_{\Gamma}\ \gamma_0\ \gamma_1$ and an equivalence $\eta^{T} : \ldots \simeq \mathsf{BridgeP}_{x.T(\gamma\gamma\ x)}\ (t_0 : T\ \gamma_0)(t_1 : T\ \gamma_1)$ where $\gamma\gamma : \mathsf{Bridge}_{\Gamma}\ \gamma_0\ \gamma_1$. This is explained in Section 3.1 and illustrated on an example in Section 3.1.4.

We call this informal sufficient condition for obtaining free theorems, which asks that all (dependent) types are equipped with a characterization of their Bridge/BridgeP type as logical relations, the *structure relatedness principle* (SRP). The SRP is precisely stated in Section 3.1. The reason behind this name is that there exists an analogous principle asking instead that all (dependent) types feature a characterization of their ≡/PathP types as types of isomorphisms. The latter principle is known (to varying degrees of generality, see Section 6.3) as the *structure identity principle* (SIP) in HoTT/UF.

The second improvement we make compared to low-level proofs stems from the observation that proving the SRP or the SIP "by hand" at a given type can quickly get intractable, as explained in Section 3.2. To remedy this situation we introduce in Section 3.3 a shallowly embedded domain-specific language (DSL) implemented as an Agda --bridges library that allow the user to (1) show the SRP at a type T by merely writing their type T in the DSL (using the rules in Section 3.3.1) and (2) derive free theorems for T in a straightforward manner (see the param theorem of Section 3.3.2). We call our DSL *relational observational type theory* (ROTT). By contrast with low-level proofs, ROTT provides abstractions to write user-friendly, modular and concise proofs.

3.1 The SRP and Bare Parametricity

The first improvement we make for better internal parametricity proofs is to systematically factor proofs of free theorems into two simpler statements: the structure relatedness principle (SRP) on one side, and bare parametricity on the other. We first explain these principles and then illustrate their use by deriving the global free theorem (4) of Section 1 in Agda --bridges.

3.1.1 Bare Parametricity. Bare parametricity is simply the fact that all programs defined in Agda --bridges have a canonical action on bridges. It can be summarized as the following bare-param program:

```
bare-param : ∀ {Γ : Type} {T : Γ → Type} (p : ∀ γ → T γ) (γ₀ γ₁ : Γ)
  (γγ : Bridge γ₀ γ₁) → BridgeP (λ x → T (γγ x)) (p γ₀) (p γ₁)
bare-param p γ₀ γ₁ γγ = λ x → p (γγ x)
```

3.1.2 The SRP, Relativistic Reflexive Graphs and the SIP. The structure relatedness principle (SRP) is the following metatheoretical principle:

```
record RRGraph : Type₁ where
  field
    cr : Type
    lrel : cr → cr → Type
    requ : ∀ (a b : cr) → lrel a b ≃ Bridge a b
open RRGraph public

record DispRRG (Γ : RRGraph) : Type₁ where
  field
    dcr : Γ .cr → Type
    dlrel : ∀ (γ₀ γ₁ : Γ .cr) (γr : Γ .lrel γ₀ γ₁) (a₀ : dcr γ₀) (a₁ : dcr γ₁) → Type
    drequ : (γ₀ γ₁ : Γ .cr) (γr : Γ .lrel γ₀ γ₁) (γγ : Bridge γ₀ γ₁) (γprf : γr [ (Γ .requ γ₀ γ₁) ] γγ)
      (a₀ : dcr γ₀) (a₁ : dcr γ₁) → dlrel γ₀ γ₁ γr a₀ a₁ ≃ BridgeP (λ x → dcr (γγ x)) a₀ a₁
open DispRRG public

→Form : ∀ {Γ : RRGraph} (A B : DispRRG Γ) → DispRRG Γ
→Form A B .dcr γ = (A .dcr γ → B . dcr γ)
→Form A B .dlrel γ₀ γ₁ γr f₀ f₁ = ∀ a₀ a₁ → (ar : A .dlrel γ₀ γ₁ γr a₀ a₁) → B .dlrel γ₀ γ₁ γr (f₀ a₀) (f₁ a₁)
→Form A B .drequ γ₀ γ₁ γr γγ γprf f₀ f₁ = flip compEquiv extentEquiv --under the hood: extent
  ((equivΠCod λ a₀ → equivΠCod λ a₁ →
  equivΠ' (A .drequ γ₀ γ₁ γr γγ γprf a₀ a₁) λ {ar} {aa} aprf →
  B .drequ γ₀ γ₁ γr γγ γprf (f₀ a₀) (f₁ a₁)))
```

Fig. 3. (Displayed) relativistic reflexive graphs in Agda --bridges, and their →FM semantic rule.

Structure Relatedness Principle (SRP): For each type Γ : Type of Agda --bridges, (resp. type family $T : \Gamma \to$ Type) the Bridge type of Γ (resp. the BridgeP type of T) can be characterized as a type of logical relations (resp. heterogeneous logical relations).

In other words, the SRP conveys the idea that bridges act as logical relations *at all types**. For instance the SRP holds at Type thanks to the relativity theorem of Section 2.5, and the SRP holds at function types $(a : A) \to B$ thanks to the extentEquiv theorem of Section 2.4.

Contrary to bare parametricity, the SRP can not be stated as an object-level theorem of Agda --bridges. The reason is that types of logical relations are ad hoc: there is a priori no Agda --bridges function Lrel : Type → Type acting as an internal parametricity translation, computing for each Γ its type of logical relations Lrel Γ. Indeed parametricity translations are defined by induction on the syntax, so using a type-casing operation. But having a first-class type-casing operator on types would contradict the internal parametricity of Agda --bridges! This means that the quantification (*) must be stated metatheoretically.

Since the SRP can not be obtained as a theorem, we package it as a definition on types (or type families). For reasons explained hereafter, types that satisfy the SRP are called *relativistic reflexive graphs* (RRG). Moreover type families $T : \Gamma \to$ Type satisfying the SRP are called *displayed relativistic reflexive graphs* (dRRG). In what follows RRGs and dRRGs are defined in plain English. The corresponding formal Agda --bridges definitions of these structures are provided in Fig. 3.

Recall that we use a postfix r notation to denote (potentially heterogeneous) logical relations ar and a double-letter notation aa to denote (potentially dependent) bridges. Additionally, note that we use the notation $a_0\ [e]\ a_1$ to signify that e .fst $a_0 \equiv_{A_1} a_1$ where $e : A_0 \simeq A_1$ and $a_0 : A_0, a_1 : A_1$.

Definition 3.1 (Relativistic reflexive graph (RRG)). A relativistic reflexive graph is a type Γ : Type which, for all elements $\gamma_0, \gamma_1 : \Gamma$, is equipped with a type denoted $\Gamma\{\gamma_0, \gamma_1\}$: Type and an equivalence denoted $\eta^\Gamma : \Gamma\{\gamma_0, \gamma_1\} \simeq \text{Bridge}_\Gamma\ \gamma_0\ \gamma_1$.

Members of $\Gamma\{\gamma_0, \gamma_1\}$ are called logical relations at Γ between γ_0, γ_1. The η equivalence is called the relativistic equivalence of Γ. When the context is clear, we allow ourselves to refer to the RRG given by the triple $(\Gamma, \lambda\gamma_0\,\gamma_1 \to \Gamma\{\gamma_0, \gamma_1\}, \lambda\gamma_0\,\gamma_1 \to \eta^\Gamma)$ simply as Γ. Next, the notion of displayed relativistic reflexive graph (dRRG) is an indexed version of the above notion. The definition is not straightforward but exactly encodes what we want: types that carry characterizations of their BridgeP types. To help the reader parse the definition, the indexed counterpart of each RRG operation is written in **bold** font.

Definition 3.2 (Displayed relativistic reflexive graph over − (dRRG over −)). Let Γ be a RRG. A displayed relativistic reflexive graph T over Γ is

- For every $\gamma : \Gamma$ **a type denoted** T_γ with...
- For every $\gamma_0\,\gamma_1\,(\gamma r : \Gamma\{\gamma_0, \gamma_1\})(t_0 : T\,\gamma_0)(t_1 : T\,\gamma_1)$ **a type denoted** $T\{t_0, t_1\}_{\gamma r}$ with ...
- For every $(\gamma r : \Gamma\{\gamma_0, \gamma_1\})$ and $(\gamma\gamma : \mathsf{Bridge}_\Gamma\,\gamma_0\,\gamma_1)$ such that $\gamma r\,[\eta^\Gamma]\,\gamma\gamma$ and for every $(t_0 : T\,\gamma_0), (t_1 : T\,\gamma_1)$, **an equivalence denoted** $\eta^T : T\{t_0, t_1\}_{\gamma r} \simeq \mathsf{BridgeP}_{x.T(\gamma\gamma\,x)}\,t_0\,t_1$.

Members of $T\{t_0, t_1\}_{\gamma r}$ are called heterogeneous logical relations at T between t_0, t_1 over the logical relation γr. If the triple $(T,\ \lambda\ldots \to T\{t_0, t_1\}_{\gamma r}\ ,\ \lambda\ldots \to \eta^T)$ is a dRRG over Γ, we allow ourselves to refer to it simply as T. In that case, we also adopt the suggestive notation $(\gamma : \Gamma) \vDash T_\gamma$ dRRG by analogy with the *type* judgment of type theory $(x_0 : A_0, \ldots, x_n : A_n \vdash T\ \mathsf{type})$.

We motivate our terminology of RRGs and dRRGs for types that satisfy the SRP. First, types Γ that have the SRP, i.e., are equipped with types $\Gamma\{\gamma_0, \gamma_1\}$ and an equivalence $\eta : \Gamma\{\gamma_0, \gamma_1\} \simeq \mathsf{Bridge}_\Gamma\,\gamma_0\,\gamma_1$, are reflexive graphs. Indeed we can regard the logical relations of Γ as its edges and we can pick $\eta^{-1}(\lambda x.\,\gamma) : \Gamma\{\gamma, \gamma\}$ for its reflexivity edges. Second, the archetypal type satisfying the SRP is Type, thanks to relativity. Hence types that satisfy the SRP are called *relativistic reflexive graphs*. Following Ahrens and Lumsdaine's [2019] convention for naming dependent mathematical objects, we call the dependent version of a RRG a displayed RRG.

By exact analogy with the SRP, the structure identity principle (SIP; see Section 6.3 for references) is a metatheoretical statement asking that, for each type Γ (resp. $T : \Gamma \to T$) the $\equiv$ type of Γ (c.f. bridge type; resp. PathP type of T) can be characterized as a type of *isomorphisms* (c.f. logical relations). That is to say, the SIP conveys the idea that paths act as isomorphisms at all types. For instance the SIP holds at Type and at function types $(a : A) \to B$ thanks to the univalence and funextEquiv theorems of Section 2.1.3. Types satisfying the SIP are most commonly known as univalent groupoids, or setoids in the literature (c.f. relativistic reflexive graphs).

3.1.3 Examples of RRGs and dRRGs. We give two examples of RRGs and one example of dRRG. The upshot is that each (dependent) relational extensionality principle that we can prove (see Section 2.6) for a potentially composite type, can be repackaged as a (d)RRG.

First, we define the composite type of premonoids $\mathsf{PreMon} = \Sigma[\,M \in \mathsf{Type}\,]\,M \times (M \to M \to M)$. This type can be turned into an RRG. Indeed we can pick $\Gamma = \mathsf{PreMon}$ and define $\mathsf{PreMon}\{M_0, M_1\}$ as the type of (actual) logical relations of premonoids between M_0, M_1. A logical relation of premonoids is a relation between M_0, M_1 with a proof that the neutral elements of M_0, M_1 are related, and a proof that the binary functions are pointwise related. By combining the principles for $\Sigma, \times, \mathsf{Type}, \to$ appearing in Section 2.6, one can prove that the type $\mathsf{PreMon}\{M_0, M_1\}$ is equivalent to $\mathsf{Bridge}_{\mathsf{PreMon}}\,M_0\,M_1$ which makes $\Gamma = \mathsf{PreMon}$ an RRG. Second, the type $\mathsf{Bool} \to \mathsf{Bool}$ can be turned into an RRG. We can indeed pick $(\mathsf{Bool} \to \mathsf{Bool})\{f_0, f_1\} = (\forall b_0\,b_1 \to f_0\,b_0 \equiv_{\mathsf{Bool}} f_1\,b_1)$ and use the appropriate relational extensionality principles, including the one for Bool mentioned in (2.6) to build the expected equivalence. Our third example regards the List type former. Recall that the triple $(\mathsf{Type}, \lambda A_0\,A_1 \to A_0 \to A_1 \to \mathsf{Type}, \mathsf{relativity})$ turns Type into a RRG. We assert that the type

family $\lambda X \to \mathsf{List}\, X$ is a displayed RRG over the Type RRG, i.e., $(X : \mathsf{Type}) \vDash \mathsf{List}\, X$ dRRG. The reason is that we can pick $(\lambda X \to \mathsf{List}\, X)\{as_0, as_1\}_R = \mathsf{ListRel}\, R\, as_0\, as_1$, the inductive characterization of $\mathsf{BridgeP}_{x.\,\mathsf{List}(AA\,x)}$ discussed in Section 2.6.

3.1.4 SRP + Bare Parametricity. Next, we illustrate how the SRP and bare parametricity can be used to improve proofs of internal free theorems. The main idea is that since (1) bare parametricity tells us that programs act on bridges and (2) the SRP guarantees that bridges uniquely correspond to logical relations, we expect that (3) programs act on logical relations as well. And all free theorems are consequences of the latter fact. We derive the global free theorem (4) of Section 1 in Agda --bridges, or rather a slightly reduced version of it for sparing space.

$$\begin{aligned}&\mathsf{fthm} : \forall\, \{A_0\ A_1 : \mathsf{Type}\}\ (f : A_0 \to A_1)\ (as_0 : \mathsf{List}\ A_0)\\ &\quad (p : (X : \mathsf{Type}) \to \mathsf{List}\ X \to \mathsf{List}\ X) \to \mathsf{map}\ f\ (p\ A_0\ as_0) \equiv p\ A_1\ (\mathsf{map}\ f\ as_0)\end{aligned}$$

The first step of our proof is to derive the SRP for the domain and codomain of p. In other words we must (1) equip Type with an RRG structure (we chose the one induced by relativity) and (2) equip the type family $\lambda X.\, \mathsf{List}\, X \to \mathsf{List}\, X$ with a dRRG structure over the Type RRG. This amounts to proving the following characterization of the BridgeP type of $\lambda X.\, \mathsf{List}\, X \to \mathsf{List}\, X$ where $A_0, A_1 : \mathsf{Type}$, $R : A_0 \to A_1 \to \mathsf{Type}$, $AA : \mathsf{Bridge}\, A_0\, A_1$, $Aprf : R\, [\, \mathsf{relativity}\,]\, AA$, and $q_\varepsilon : \mathsf{List}\, A_\varepsilon \to \mathsf{List}\, A_\varepsilon$ for $\varepsilon = 0, 1$. The proof uses the relational extensionality principles extentEquiv, ListvsBridgeP and the one for Gel (see Section 2.5.1, 2.6) as well as the fact that all type formers preserve equivalences. In the spirit of parametricity translations, an appropriate relational extensionality principle is used based on the head of the type former appearing in the BridgeP type at hand.

$$\begin{aligned}&\mathsf{BridgeP}_{x.\,\mathsf{List}(AA\,x)\to\mathsf{List}(AA\,x)}\ q_0\ q_1 \quad \simeq\\ &\forall as_0\ as_1 \to \mathsf{BridgeP}_{x.\mathsf{List}\,AA\,x}\ as_0\ as_1 \to \mathsf{BridgeP}_{x.\,\mathsf{List}(AA\,x)}\ (q_0\ as_0)(q_1\ as_1) \quad \simeq\\ &\forall as_0\ as_1 \to (\mathsf{ListRel}\ (\mathsf{BridgeP}_{x.\,AA\,x})\ as_0\ as_1) \to (\mathsf{ListRel}\ (\mathsf{BridgeP}_{x.\,AA\,x})\ (q_0\ as_0)\ (q_1\ as_1)) \simeq\\ &\forall as_0\ as_1 \to (\mathsf{ListRel}\, R\, as_0\, as_1) \to (\mathsf{ListRel}\, R\, (q_0\ as_0)\ (q_1\ as_1))\end{aligned}$$

The second step of our proof is to "conjugate" bare-param with the SRP proof obligations we produced for Type and $\lambda X.\, \mathsf{List}\, X \to \mathsf{List}\, X$. Let A_0, A_1, f, as_0, p be as in fthm. We first convert the graph relation of f denoted $\mathsf{Gr}\, f = \lambda\, a_0\, a_1 \to f\, a_0 \equiv_{A_1} a_1$ into a bridge denoted $AA : \mathsf{Bridge}_{\mathsf{Type}}\, A_0\, A_1$, using relativity. Then we apply bare parametricity to AA.

$$\mathsf{bare\text{-}param}\ \{T = \lambda\, X \to \mathsf{List}\, X \to \mathsf{List}\, X\}\ p\, A_0\, A_1\, AA\ :\ \mathsf{BridgeP}_{x.\,\mathsf{List}(AA\,x)\to\mathsf{List}(AA\,x)}\ (p\, A_0)(p\, A_1)$$

Finally we use the above dependent principle for $\lambda X.\, \mathsf{List}\, X \to \mathsf{List}\, X$ and obtain a proof $\mathsf{pf} : \forall as_0\, as_1 \to (\mathsf{ListRel}\, (\mathsf{Gr}\, f)\, as_0\, as_1) \to (\mathsf{ListRel}\, (\mathsf{Gr}\, f)\, (p\, A_0\, as_0)\, (p\, A_1\, as_1))$. By a simple list induction we see that $\mathsf{ListRel}\, (\mathsf{Gr}\, f)$ is equal to the relation $\lambda\, as_0\, as_1 \to (\mathsf{map}\, f)\, as_0 \equiv as_1$. Thus $\mathsf{pf}\ as_0\ (\mathsf{map}\ f\ as_0)\ \mathsf{refl}$ grants the free theorem fthm.

The technique of factoring free theorems into SRP proof obligations and a call to bare-param is an improvement compared to low-level parametricity proofs like the one presented in Section 2.7. The proofs are conceptually easier. Moreover they allow for more compositionality. Indeed, contrary to Section 2.7 we are now in position to reuse the SRP proof obligations obtained for Type and $\lambda X.\, \mathsf{List}\, X \to \mathsf{List}\, X$ to derive (1) other free theorems for programs $p : (X : \mathsf{Type}) \to \mathsf{List}\, X \to \mathsf{List}\, X$ and even (2) shorter proofs of free theorems for a composite type having $\mathsf{List} - \to \mathsf{List} -$ as a subexpression. However it turns out that proving SRP obligations "by hand" like in the above is challenging, in fact strictly more challenging than SIP obligations, as explained in the next subsection.

3.2 Obstructions to the SRP and SIP

In this subsection we argue that proving the SIP, or worse, the SRP at a given type can get tedious. We begin with an example of SIP and SRP proofs for the type of pointed unary type operations $\mathsf{PointedOp} = \Sigma[\, F \in \mathsf{Type} \to \mathsf{Type} \,]\ ((X : \mathsf{Type}) \to X \to F\,X)$.

Example 3.3 (The SIP via extensionality principles). By repeatedly applying extensionality principles (see Section 2.1.3) we can characterize the meaning of a path between such pointed operations:

$$
\begin{aligned}
&(F_0, f_0) \equiv_{\mathsf{PointedOp}} (F_1, f_1)\\
&\simeq \Sigma[\, FF \in F_0 \equiv_{\mathsf{Type}\to\mathsf{Type}} F_1 \,]\ \mathsf{PathP}_{i.(X:\mathsf{Type})\to X\to FF\,i\,X}\ f_0\ f_1\\
&\simeq \Sigma[\, FF' \in (X : \mathsf{Type}) \to (F_0\,X) \equiv_{\mathsf{Type}} (F_1\,X) \,]\ \mathsf{PathP}_{i.(X:\mathsf{Type})\to X\to FF'\,X\,i}\ f_0\ f_1\\
&\simeq \Sigma[\, e \in (X : \mathsf{Type}) \to F_0\,X \simeq F_1\,X \,]\ \mathsf{PathP}_{i.(X:\mathsf{Type})\to X\to \mathsf{ua}\,(e\,X)\,i}\ f_0\ f_1\\
&\simeq \Sigma[\, e \in (X : \mathsf{Type}) \to F_0\,X \simeq F_1\,X \,]\ \big((X : \mathsf{Type}) \to (x : X) \to \mathsf{PathP}_{i.\mathsf{ua}\,(e\,X)\,i}\ (f_0\,X\,x)\ (f_1\,X\,x)\big)\\
&\simeq \Sigma[\, e \in (X : \mathsf{Type}) \to F_0\,X \simeq F_1\,X \,]\ ((X : \mathsf{Type}) \to (x : X) \to f_0\,X\,x\ [e\,X]\ f_1\,X\,x)
\end{aligned}
$$

We conclude that a path between pointed operations (F_0, f_0) and (F_1, f_1) consists of a pointwise equivalence between F_0 and F_1, compatible with the pointings f_0 and f_1.

Example 3.4 (The SRP via relational extensionality principles). Set $\mathsf{Rel}\,X_0\,X_1 := X_0 \to X_1 \to \mathsf{Type}$ and $\mathsf{ra} := \mathsf{relativity}$. We can characterize bridges at $\mathsf{PointedOp}$ by applying the principles discussed in Section 2.5.1, 2.6.

$$
\begin{aligned}
&\mathsf{Bridge}_{\mathsf{PointedOp}}\ (F_0, f_0)\ (F_1, f_1)\\
&\simeq \Sigma[\, FF : \mathsf{Bridge}_{\mathsf{Type}\to\mathsf{Type}}\ F_0\ F_1 \,]\ \mathsf{BridgeP}_{y.(X:\mathsf{Type})\to X\to FF\,y\,X}\ f_0\ f_1\\
&\simeq \Sigma[\, FF' : (X_0\,X_1 : \mathsf{Type}) \to \mathsf{Bridge}_{\mathsf{Type}}\,X_0\,X_1 \to \mathsf{Bridge}_{\mathsf{Type}}\,(F_0\,X_0)\,(F_1\,X_1) \,]\\
&\qquad (X_0\,X_1 : \mathsf{Type})(XX : \mathsf{Bridge}_{\mathsf{Type}}\,X_0\,X_1)(x_0 : X_0)(x_1 : X_1) \to\\
&\qquad \mathsf{BridgeP}_{y.\,XX\,y}\,x_0\,x_1 \to \mathsf{BridgeP}_{y.FF'\,X_0\,X_1\,XX\,y}\ (f_0\,X_0\,x_0)\ (f_1\,X_1\,x_1)\\
&\simeq \Sigma[\, Fr : (X_0\,X_1 : \mathsf{Type}) \to \mathsf{Rel}\,X_0\,X_1 \to \mathsf{Rel}\,(F_0\,X_0)\,(F_1\,X_1) \,]\\
&\qquad (X_0\,X_1 : \mathsf{Type})(R : \mathsf{Rel}\,X\,Y)(x_0 : X_0)(x_1 : X_1) \to\\
&\qquad \mathsf{BridgeP}_{y.\mathsf{ra}\,R\,y}\,x_0\,x_1 \to \mathsf{BridgeP}_{y.\mathsf{ra}\,(Fr\,X_0\,X_1\,R)\,y}\ (f_0\,X_0\,x_0)\ (f_1\,X_1\,x_1)\\
&\simeq \Sigma[\, Fr : (X_0\,X_1 : \mathsf{Type}) \to \mathsf{Rel}\,X_0\,X_1 \to \mathsf{Rel}\,(F_0\,X_0)\,(F_1\,X_1) \,]\\
&\qquad (X_0\,X_1 : \mathsf{Type})(R : \mathsf{Rel}\,X\,Y)(x_0 : X_0)(x_1 : X_1) \to\\
&\qquad (R\,x_0\,x_1) \to Fr\,X_0\,X_1\,R\,(f_0\,X_0\,x_0)(f_1\,X_1\,x_1)
\end{aligned}
$$

We conclude that a bridge between pointed operations (F_0, f_0) and (F_1, f_1) consists of a relation transformer Fr between them such that the pointings f_0 and f_1 send related pairs to related pairs.

The examples above required rote work: we had to apply a series of extensionality principles which was entirely dictated by the formation of $\mathsf{PointedOp}$. Hence, it is clear that proofs of the SIP and SRP at a type T scale at least with the complexity of T: for each type former F used to define T, an appropriate extensionality principle must be used to swap PathP/$\mathsf{BridgeP}$ and F.

Moreover we argue that proving the SRP is in general strictly harder than proving the SIP, because of three obstructions which we list here.

The first obstruction is that the $\mathsf{extentEquiv}$ principle of Section 2.4 always produces an extra Bridge type when characterizing the type of bridges between two given functions. Both generated Bridge types must be further characterized to finish the SRP proof at hand. This is to compare with the $\mathsf{funextEquiv}$ principle of Section 2.1.3 which (if not in the fully dependent case) does not produce an extra PathP type.

The second and third obstructions relate to the fact that, in the SIP case, some tools are available to dismiss part of the rote work seen above. A first such tool is a theorem (see Rijke [2022] e.g.) that can be seen as a reformulation in cubical type theory of the J rule of equality types. Recall that a type A is contractible (isContr A) if it is equivalent to the unit type $\{*\}$.

Theorem 3.5 (Fundamental theorem of identity types). *Assume $A : \mathit{Type}$ and a relation $Eq : A \to A \to \mathit{Type}$. Suppose that Eq is reflexive, i.e., $\forall a \to Eq\, a\, a$. Additionally, suppose that $\forall a_0 \to \mathit{isContr}\,(\Sigma[\, a_1 \in A\,]\, Eq\, a_0\, a_1)$. Then $\forall a_0\, a_1 \to (Eq\, a_0\, a_1) \simeq (a_0 \equiv_A a_1)$.*

This theorem might allow us to shortcut proofs as in Example 3.3, especially in the case of ordinary data types, by simply proving that the end result satisfies the above criteria. However, since bridges have no elimination principles like J, the theorem does not to our knowledge translate to the relational setting.

Similarly, the third obstruction is the lack of the following result, standard in HoTT but only available in case of the SIP. Recall that a type P is an h-proposition if any two inhabitants of P are equal, i.e., $\mathrm{isProp}\, P = \forall p_0\, p_1 \to p_0 \equiv_P p_1$. For instance the empty and unit types are h-propositions. The theorem guarantees the existence and unicity of heterogeneous paths between proofs of h-propositions.

Theorem 3.6 (Heterogenous equality of proofs). *Let $P : I \to \mathit{Type}$ be a line of h-propositions, i.e., there is a map $isp : (i : I) \to \mathit{isProp}\,(P\, i)$. Suppose that $p_0 : P\, i0$ and that $p_1 : P\, i1$. Then $\mathit{isContr}\,(\mathit{PathP}_{i.\,P\, i}\, p_0\, p_1)$*[5].

Typically, this theorem is used while proving the SIP for structured types of the form $\Sigma[\, a \in A\,]\, (P\, a)$ where for all a, $P\, a$ is an h-proposition. For instance this is the reason why two equivalences $e_0, e_1 : A_0 \simeq A_1$ are equal if and only if their underlying functions are equal $e_0 \equiv e_1 \simeq (e_0.\mathrm{fst} \equiv e_0.\mathrm{fst})$. This is to compare with the relational extensionality principle for equivalences evoked in Section 2.5.2 which has the hardest proof amongst basic relational principles.

Because of these obstructions to SRP proofs, we have developed a domain-specific language (DSL) to obtain proofs of the SRP at a given type T by merely writing the type T in the DSL. Our DSL is explained in the next subsection.

3.3 ROTT

The second improvement we make compared to low-level proofs of free theorems is the introduction of *relational observational type theory* (ROTT): a domain-specific language (DSL) implemented as a library in Agda --bridges. It allows users to (1) show the SRP at a type T by merely writing their type T in the DSL and (2) derive free theorems for T in a straightforward manner.

Since ROTT has abstractions that compose *dependent types* T satisfying the SRP (i.e. dRRGs from Definition 3.2), it is itself a dependent type theory. To be more precise, ROTT is a type theory *shallowly embedded* in Agda --bridges. This means that ROTT is not defined as some kind of data type of expressions whose constructors would be inference rules used to write types T. Instead, ROTT is an Agda --bridges library (part of our accompanying library) comprised of a bunch of theorems called "semantic rules" explaining how to compose dRRGs. These Agda --bridges theorems are discussed in Section 3.3.1. As an example, we program in Fig. 3 the $\to_{\mathrm{FM}}$ semantic rule of ROTT directly in Agda --bridges.

Additionally, ROTT also features a param theorem letting the user straightforwardly deduce free theorems from appropriate (d)RRGs instances (obtained with ROTT, e.g.). We see param as one of the rules of ROTT. It is discussed in Section 3.3.2.

[5]For bridges, only $\mathrm{isProp}\,(\mathrm{BridgeP}_{x.\,P\, x}\, p_0\, p_1)$ holds. By (2.5.1), $\mathrm{BridgeP}_{x.\,\mathrm{Gel}\, P_0\, P_1\, (\lambda -\, -.\, \bot)\, x}\, p_0\, p_1 \simeq \bot$.

$$\frac{\Gamma \text{ RRG} \qquad \Gamma \vDash T \text{ dRRG}}{\Gamma \# T \text{ RRG}}\ \text{CTX-EXT} \qquad \frac{\Gamma \vDash A \text{ dRRG} \qquad \Gamma \# A \vDash B \text{ dRRG}}{\Gamma \vDash \Pi\text{fm}\, A\, B \text{ dRRG}}\ \Pi\text{FM}$$

$$\frac{\Gamma \vDash A \text{ dRRG} \qquad \Gamma \# A \vDash B \text{ dRRG}}{\Gamma \vDash \Sigma\text{fm}\, A\, B \text{ dRRG}}\ \Sigma\text{FM} \qquad \frac{\Gamma \vDash f : \Pi\text{fm}\, A\, B \qquad \Gamma \vDash a : A}{\Gamma \vDash \text{app}\, f\, a : B[a]}\ \text{APP}$$

$$\frac{\Gamma \text{ NRG}}{\Gamma \vDash \text{Tyfm dNRG}}\ \text{TYFM} \qquad \frac{\Gamma \vDash A \text{ dRRG} \qquad \Gamma \vDash a_0 : A \qquad \Gamma \vDash a_1 : A}{\Gamma \vDash \equiv\text{fm}\, a_0\, a_1 \text{ dRRG}}\ \equiv\text{FM}$$

Fig. 4. Some rules of ROTT. Some premises are not displayed.

3.3.1 The Standard Rules of ROTT. The key idea behind ROTT is to remark that RRGs Γ act as if they were *contexts* of a certain type theory, and displayed RRGs T over Γ act as if they were *types* of a type theory. In other words it seems like the type of all RRGs RRGraph somehow constitutes a model of type theory. For the expert reader this can be summarized as the fact that ROTT is a raw category-with-families (CwF) structure (with support for various type formers) on the category formed by relativistic reflexive graphs and their morphisms[6]. Here "raw" means that no equations between the CwF operations are required. There is no need to define what a (raw) CwF is since we only deal with one instance here and since in practice our proofs of the SRP sometimes combine manual reasoning with using the ROTT interface.

ROTT features standard (shallowly embedded!) type theoretic rules to combine RRGs and dRRGs, treating them as type-theoretic contexts and types in contexts, respectively. Some of those rules appear in Fig. 4 using traditional type-theoretic syntax. Recall that our notation for displayed RRGs over Γ is $\Gamma \vDash T$ dRRG. Each of those rules is an Agda --bridges program parametrized by the premises appearing in the rule. For example, the $\to$FM rule of ROTT (a restricted version of ΠFM) is programmed directly in Agda --bridges in Fig. 3.

Let us explain how some of those rules are defined in Agda --bridges. The important idea here is that those rules compose the relational extensionality principles of their premises to yield composite types satisfying the SRP. This means that from the point of view of the user of ROTT, the SRP is proved by the system while types are being written with those rules. As can be expected, given Γ a RRG and T a displayed RRG over it, the context extension $\Gamma \# T$ is a RRG whose carrier is simply $\Sigma[\, \gamma \in \Gamma \,]\, T_\gamma$. Its type of logical relations $(\Gamma \# T)\{(\gamma_0, t_0), (\gamma_1, t_1)\}$ is moreover defined as $\Sigma[\, \gamma r \in \Gamma\{\gamma_0, \gamma_1\}]\, T\{t_0, t_1\}_{\gamma r}$. The fact that this type matches the corresponding bridge type $\text{Bridge}_{\Gamma \# T}\ (\gamma_0, t_0)\ (\gamma_1, t_1)$, can easily be deduced from the same fact for Γ and T. Regarding Πfm and Σfm, and supposing that Γ is empty, $\Pi\text{fm}\, A\, B$ and $\Sigma\text{fm}\, A\, B$ are RRGs[7] with carriers $(a : A) \to B\, a$ and $\Sigma[\, a \in A \,]\, (B\, a)$. In particular $((a : A) \to B\, a)\{f_0, f_1\}$ is defined as the type $\forall\, a_0\, a_1 (ar : A\{a_0, a_1\}) \to B\{f_0\, a_0, f_1\, a_1\}_{ar}$ and this type can again be proven to match the corresponding Bridge type by composing the relativistic equivalences of A and B. The Tyfm rule equips Type with a dRRG structure in the expected way. For an empty Γ, the rule sets $\text{Type}\{A_0, A_1\} = A_0 \to A_1 \to \text{Type}$ and uses relativity. Finally ROTT also features a term judgment written $\Gamma \vDash a : A$ needed because arbitrary dependent types might mention terms in their definition. We do not state the definition of this judgment and merely indicate that $\Gamma \vDash a : A$ holds if $a : (\gamma : \Gamma) \to A_\gamma$ features a custom action on logical relations $\gamma r : \Gamma\{\gamma_0, \gamma_1\}$ which matches it canonical action on $\text{Bridge}_\Gamma\ \gamma_0\ \gamma_1$, i.e., $\text{bare-param}\ a\ \gamma_0\ \gamma_1$.

3.3.2 Internal Observational Parametricity. We state the param semantic rule of ROTT. Once again we do so as if it was a syntactic type-theoretic rule even though param really is an Agda --bridges program parametrized by the premises appearing in the rule. The theorem states that, given an

[6]In our development those morphisms are called relativistic, or native relators but we will not need this notion here.
[7]There is a slight mismatch between dRRGs in empty contexts and RRGs, but we ignore it here.

RRG Γ and a displayed RRG T over it, all external dependent functions (i.e. all functions definable in Agda --bridges) from Γ to T respect logical relations.

$$\frac{\Gamma \text{ RRG} \qquad \Gamma \vDash T \text{ dRRG} \qquad p : (\gamma : \Gamma) \to T_\gamma \qquad \gamma_0, \gamma_1 : \Gamma \qquad \gamma r : \Gamma\{\gamma_0, \gamma_1\}}{\text{param } \Gamma\ T\ p\ \gamma_0\ \gamma_1\ \gamma r\ :\ T\{p\,\gamma_0, p\,\gamma_1\}_{\gamma r}}\ \text{PARAM}$$

The proof of this theorem is elementary and proceeds in three steps, similar to the proof appearing in Section 3.1.4. We omit to write .fst to extract direct maps out of equivalences. First convert the logical relation γr into a bridge $\eta^\Gamma\ \gamma r : \text{Bridge}_\Gamma\ \gamma_0\ \gamma_1$. Second use bare parametricity to obtain bare-param $p\ \gamma_0\ \gamma_1\ (\eta^\Gamma\ \gamma r)$ which has type $\text{BridgeP}_{x.T(\eta^\Gamma\ \gamma r\ x)}\ (p\,\gamma_0)(p\,\gamma_1)$. Third by definition we know that $\gamma r\ [\eta^\Gamma]\ (\eta^\Gamma\ \gamma r)$. Hence we may use the relativistic equivalence η^T of T to obtain ${\eta^T}^{-1}$(bare-param $p\ \gamma_0\ \gamma_1\ (\eta^\Gamma\ \gamma r)$) which has type $T\{p\,\gamma_0, p\,\gamma_1\}_{\gamma r}$. This concludes the proof and definition of param.

The param theorem draws inspiration from observational type theory. It can indeed be seen as a relational analogue of the *ap* inference rule (see e.g. [Altenkirch et al. 2022] and Section 6) which states that terms of observational type theory act on identifications or isomorphisms, that is, observational proofs of equality. For this reason we say that our internal param-etricity theorem is also observational. In the next section we use ROTT and its param rule to obtain modular and concise proofs of internal free theorems.

4 INTERNAL OBSERVATIONAL PARAMETRICITY APPLIED

In this section we obtain several free theorems as one-liner invocations of the param theorem of Section 3.3.2. This is done by first constructing appropriate (d)RRGs using the rules of ROTT. All of our examples can be consulted in our accompanying library.

4.1 Reproving fthm and lowChurchBool

We begin by recasting our proof of the global free theorem fthm from Section 3.1.4, this time using ROTT and its param rule. Let $p : (X : \text{Type}) \to \text{List}\,X \to \text{List}\,X$, let $f : A_0 \to A_1$ for A_0, A_1 two types and let $as : \text{List}\,A_0$. We want to apply param at program p and at (logical) relation $\text{Gr}\,f : A_0 \to A_1 \to \text{Type}$. In order to do so we must first provide an RRG structure for Type, the domain of p. This is achieved using the Tyfm rule of ROTT. Second, we must prove that $X : \text{Type} \vDash \text{List}\,X \to \text{List}\,X$ dRRG. By the Πfm rule of ROTT (or rather $\to$FM of Fig. 3) it is sufficient to prove (twice) that $X : \text{Type} \vDash \text{List}\,X$ dRRG. The latter displayed RRG appears as an example in Section 3.1.3. At this point all premises of param have been supplied.

Note how ROTT allows a significant improvement compared to proofs of the SRP "by hand" as shown Section 3.1.4. Instead of proving a lengthy equivalence chain, we simply have to write the following in Agda --bridges, using the →Form and X⊨ListX programs implemented by the ROTT library (Agda identifiers can feature symbols, as in X⊨ListX).

```
X⊨ListX→ListX : DispRRG TypeRRG
X⊨ListX→ListX = →Form X⊨ListX X⊨ListX
```

Looking at the conclusion of param we are about to obtain something of type $(\lambda X \to \text{List}\,X \to \text{List}\,X)\{p\,A_0, p\,A_1\}_{\text{Gr}\,f}$ and contrary to the BridgeP type former, the latter type *reduces* to the expected relational parametricity statement, i.e.:

$$\text{param TypeRRG X⊨ListX→ListX } p\ A_0\ A_1\ (\text{Gr}\,f) :$$
$$\forall xs_0\ xs_1 (xsr : \text{ListRel}\ (\text{Gr}\,f)\ xs_0\ xs_1) \to \text{ListRel}\ (\text{Gr}\,f)\ (p\,A_0\ xs_0)\ (p\,A_1\ xs_1)$$

By applying this function to $xs_0 = as_0, xs_1 = \text{map}\,f\,as_0$ and by remarking that $\text{ListRel}\ (\text{Gr}\,f)\ l_0\ l_1$ is the same predicate than $\text{map}\,f\,l_0 \equiv l_1$ we conclude the proof of fthm.

Similarly, we can reprove the lowChurchBool theorem of Section 2.7. The call to param-prf and the CH-inverse-cond module appearing in Fig. 2 are replaced by the following call to the param theorem and definition of an appropriate dRRG using ROTT:

```
param TypeRRG X⊨X→X→X k Bool A (λ b x → boolToCh b A t f ≡ x) true t refl false f refl

X⊨X→X→X : DispRRG TypeRRG
X⊨X→X→X = →Form (X⊨ElX) (→Form X⊨ElX X⊨ElX)
```

4.2 Church Encodings

Using ROTT and param we were able to prove a scheme of Church encodings for data types obtained out of a strictly positive functor, represented as a container [Abbott et al. 2005]. We first state the theorem and briefly explain its proof in Agda --bridges, and then discuss its hypotheses and significance in the more concrete case of the List type former.

Assume a type of shapes $S : \mathsf{Type}$ and a type family of positions $P : S \to \mathsf{Type}$. Assume that $S : \mathsf{Type}$ is bridge-discrete, i.e. has an equivalence $\eta^S : (s_0 \equiv s_1) \simeq \mathsf{Bridge}_S\, s_0\, s_1$. Assume that $P : S \to \mathsf{Type}$ is dependently bridge-discrete, that is, for every $s_0, s_1, (sr : s_0 \equiv s_1), (ss : \mathsf{Bridge}_S\, s_0\, s_1)$ such that $sr[\eta^S]ss$ and for every $p_0 : P\, s_0$ and $p_1 : P\, s_1$, there exists an equivalence $\eta^P : \mathsf{PathP}_{i.\,P(sr\,i)}\, p_0\, p_1 \simeq \mathsf{BridgeP}_{x.\,P(ss\,x)}\, p_0\, p_1$. Define $F : \mathsf{Type} \to \mathsf{Type}$ as $F\,X = \Sigma[\, s \in S\,]\, P\, s \to X$. Additionally define the following Agda data type $\mu\mathsf{F}$:

```
data μF : Type where
  fold : F ( μF ) → μF
```

Note here that the data type declaration is accepted by Agda since F is a container functor and its input X occurs strictly positively[8]. We assert that the following equivalence holds $\mu\mathsf{F} \simeq (X : \mathsf{Type}) \to (F\,X \to X) \to X$. The proof follows a standard pattern. Recall that $\mu\mathsf{F}$ has an elimination principle $\mu\mathsf{Frec} : \forall T \to (F\,T \to T) \to \mu\mathsf{F} \to T$. Going from left to right we can define a map toCh using the latter principle. Going from right to left we can define a map $\lambda p \to p\ \mu\mathsf{F}\ \mathsf{fold}$. Proving the first inverse condition is done by induction. The other condition requires parametricity and reads as follows (using funExt whenever needed):

$$(p : (X : \mathsf{Type}) \to (F\,X \to X) \to X)\ (A : \mathsf{Type})\ (f : F\,A \to A) \to \mathsf{toCh}(p\ \mu\mathsf{F}\ \mathsf{fold})\ A\ f \equiv_A p\ A\ f$$

This of course looks like a global free theorem in the sense of (4). We wish to obtain this equality by applying param at program p and at the (logical) relation given by the graph of the function $\mu\mathsf{Frec}\,A\,f : \mu\mathsf{F} \to A$. We denote this graph as $\mathsf{Gr}(\mu\mathsf{Frec}\,A\,f) : \mu\mathsf{F} \to A \to \mathsf{Type}$. To that end we must supply a RRG structure for the domain of p and a dRRG structure for its codomain. For its domain we use Tyfm as above. For its codomain we must show $X : \mathsf{Type} \vDash (F\,X \to X) \to X$ dRRG. Applying the Πfm rule of ROTT and other structural rules not displayed in Fig. 4 the goal is reduced to $X : \mathsf{Type} \vDash F\,X$ dRRG. Recalling that $F\,X = \Sigma[\, s \in S\,]\, P\, s \to X$ we can apply the Σfm and Πfm rules of ROTT thereby reducing the goal to providing a RRG structure for S and providing $s : S \vDash P\, s$ dRRG. We can repackage our bridge-discreteness hypotheses η^S, η^P as such structures. At this stage all premises of param have been supplied and so we expect to obtain from param something of type $(\lambda X.(F\,X \to X) \to X)\{p\ \mu\mathsf{F}, p\ A\}_{\mathsf{Gr}(\mu\mathsf{Frec}\,A\,f)}$. Again, contrary to its BridgeP

[8] Agda conveniently looks through the definition of F to decide this.

counterpart, the latter type computes to the appropriate relational parametricity statement, i.e.:

$$\mathsf{param}\ \ldots\ \mu\mathsf{F}\ A\ (\mathsf{Gr}(\mu\mathsf{Frec}\ A\ f)) :$$
$$(f_0 : F\ \mu\mathsf{F} \to \mu\mathsf{F})(f_1 : F\ A \to A)(fr : \ldots) \to \mu\mathsf{Frec}\ A\ f\ (p\ \mu\mathsf{F}\ f_0) \equiv_A\ p\ A\ f_1$$

By setting $f_0 = \mathsf{fold}, f_1 = f$ and providing an easy proof of their logical relatedness fr we get $\mu\mathsf{Frec}\ A\ f\ (p\ \mu\mathsf{F}\ \mathsf{fold}) \equiv_A\ p\ A\ f$ and this proves the theorem.

Up to some reordering lemmas, we can obtain a Church encoding for the List data type as an instance of the above scheme of Church encodings: $\mathsf{List}\ A \simeq (X : \mathsf{Type}) \to X \to (A \to X \to X) \to X$. To do this the S and P parameters of the scheme are set to $S = 1 + A$ and $P\ (\mathsf{inl}\ tt) = \bot$, $P\ (\mathsf{inr}\ a) = \mathsf{Unit}$. For this simpler List Church encoding, our bridge-discreteness hypotheses about S and P translate into the fact that A must be bridge-discrete for the encoding to hold. The reason is that, if A is not bridge-discrete, additional programs using their type variable non-parametrically will exist in the encoding. For instance Type is not bridge-discrete as its bridges are relations between types. Accordingly the encoding for $\mathsf{List}\ A$ with $A = \mathsf{Type}$ does not hold, essentially because some polymorphic programs can use their type variable non parametrically: $\lambda X\ nl\ cs.\ cs\ X\ nl : (X : \mathsf{Type}) \to X \to (\mathsf{Type} \to X \to X) \to X$. Similar considerations appear in [Nuyts and Devriese 2018].

4.3 System F

We can prove Reynolds's abstraction theorem [Reynolds 1983] for predicative System F [Leivant 1991] using ROTT and param. Indeed predicative System F is a subset of ROTT and the param theorem for dRRGs in that subset exactly expresses the abstraction theorem.

Corollary 4.1. *By analogy to Section 3.3.2, we have the following "inference rule", which states that, given a kinding context Γ of predicative System F (consisting of type variables labeled with levels) and a type T of predicative System F over this context, all external dependent functions (i.e. all functions definable in Agda --bridges) from $[\![\Gamma]\!]$ to $[\![T]\!]$ respect logical relations, where $[\![-]\!]$ is an object-level translation from System F contexts (resp. types) to RRGs (resp. dRRGs).*

$$\frac{\Gamma\ \mathit{Ctx\text{-}F} \qquad \Gamma \vdash_F T\ \mathit{type\text{-}F} \qquad p : (\gamma : [\![\Gamma]\!]) \to [\![T]\!]_\gamma \qquad \gamma_0, \gamma_1 : [\![\Gamma]\!] \qquad \gamma r : [\![\Gamma]\!]\{\gamma_0, \gamma_1\}}{\mathit{param}\ [\![\Gamma]\!]\ [\![T]\!]\ p\ \gamma_0\ \gamma_1\ \gamma r\ :\ [\![T]\!]\{p\,\gamma_0, p\,\gamma_1\}_{\gamma r}}\ \textit{PARAM-F}$$

We emphasize that this result proves parametricity of the *obvious* embedding $[\![-]\!]$ of predicative System F into Agda --bridges, which is definable in Agda --bridges. That is, $[\![-]\!]$ is defined the expected way. For instance, the System F type $X : *_0 \vdash_F X \to X$ type-F interprets as the dRRG $\mathsf{Type}_0 \vDash X \to X$ dRRG which has $\lambda X.\ X \to X$ as its carrier. In other words, the dRRG model $[\![-]\!]$ is not some contrived construction, but simply a proof that all Agda --bridges types that read as System F types satisfy the SRP.

Hence we recover Reynolds's original notion of parametricity in Agda --bridges. We remark that [Nuyts et al. 2017] could not prove this result as it lacked the power of the extent rule and therefore could not properly characterize bridges between functions. Cavallo and Harper's [2021] system (which is almost identical to Agda --bridges) could prove this result equally well but the authors did not do this, and the same holds almost certainly for Bernardy et al. [2015]. Thus, as far as we are aware, this establishes the first formal relation between traditional parametricity as defined by Reynolds for System F using a logical relation, and internally parametric type theory.

5 REIMPLEMENTING HCOMP AND TRANSP

Recall from Section 2.1 that when compared to plain dependent type theory, cubical type theories feature additional language primitives: (1) a path interval, path types, path abstraction and application, (2) Kan operations, (3) additional type formers for turning equivalences into paths

in the universe, necessary to prove univalence, (4) (optionally) higher inductive types with path constructors.

The Kan operations of cubical type theory are necessary to make paths act as well-behaved proofs of equality. For instance, these operations are used to compose paths (i.e. prove transitivity of ≡) or to turn the univalence map into an equivalence. The operational semantics of these operations is somewhat peculiar as it is defined by specifying how these operations reduce *at each type former*, i.e., by induction on the syntax of types. Hence the process of extending cubical type theory with a new type former F requires extra work: specifying how the Kan operations reduce at F.

Agda --bridges is an extension of Agda --cubical which implements CCHM cubical type theory [Cohen et al. 2017]. Thus, Agda --bridges must implement reduction clauses for the Kan operations of Agda --cubical at the BridgeP and Gel type formers. In Agda --cubical, the Kan operations are primitives named *homogeneous composition* (hcomp) and *transport* (transp).

$\mathsf{transp} : (A : \mathbb{I} \to \mathsf{Type})\ (\varphi : \mathbb{I})\ (u_0 : A\ \mathsf{i0}) \to A\ \mathsf{i1}$
$\mathsf{hcomp} : \{A : \mathsf{Type}\}\ \{\varphi : \mathbb{I}\}\ (u : (i : \mathbb{I}) \to \mathsf{Partial}\ \varphi\ A)\ (u_0 : A) \to A$

We now explain the transp (Section 5.1) and hcomp (Section 5.2) operations and how Agda --bridges extend these. The exact equations can be consulted in our implementation. Further details about transp, hcomp can be found in [Vezzosi et al. 2021]. This section ends with a brief comparison between the Kan operations implemented by Agda --bridges and those specified by the CH theory.

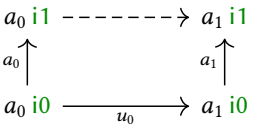
5.1 Transport

The transp operation provides a way to coerce elements of a type into elements of a path-equal type. Indeed, given $AA : A_0 \equiv_{\mathsf{Type}} A_1$ we obtain $\mathsf{transp}\ (\lambda i.\, AA\, i)\ \mathsf{i0} : A_0 \to A_1$. This map can in turn be upgraded into an equivalence and thus transp can be used to validate one direction of the univalence equivalence $A_0 \equiv A_1 \to A_0 \simeq A_1$.

Regarding the $\varphi : \mathbb{I}$ argument of transp, we merely indicate that it controls where the resulting coercion function is definitionally the identity. In other words, $\mathsf{transp}\ A\ \mathsf{i1}\ u_0$ reduces to u_0 (a typechecking side condition asks that A is constant when $\varphi = \mathsf{i1}$). "Normal" transport is recovered by setting $\varphi = \mathsf{i0}$ as illustrated above.

As explained before, the transp primitive reduces by induction on the formation of the line A, that is, a clause describing how transp reduces is specified when A is a line of Π-types, Σ-types, data types, record types, etc. Compared to --cubical, Agda --bridges implements two additional clauses for transp, handling lines of BridgeP and Gel types. For Gel, we indicate that the clause uses capturing (see Definition 2.2). We provide some details regarding the clause for BridgeP since it requires a generalized hcomp operation called mhcomp in Agda --bridges and explained hereafter.

Transporting bridges. Assume $A : \mathbb{I} \to (@x : \mathbb{BI}) \to \mathsf{Type}$ and $a_\varepsilon : (i : \mathbb{I}) \to A\, i\, \mathsf{bi}\varepsilon$. The Agda --bridges implementation adds a clause for transp specifying how the following term should reduce $\mathsf{transp}\ (i.\, \mathsf{BridgeP}_{x.\,A\,i\,x}\ (a_0\, i)\ (a_1 i))\ (\varphi : \mathbb{I})\ u_0$. This term can be represented as the dotted line in the following diagram.



Setting the result of this reduction to be the bridge $\lambda(x : \mathbb{BI}).\ \mathsf{transp}\ (i.\, A\, i\, x)\ \varphi\ (u_0\, x)$ does not work. Indeed the latter term does not have the required endpoints $a_0\ \mathsf{i1}, a_1\ \mathsf{i1}$ definitionally.

A second idea is to use the hcomp primitive (its type appears above) of --cubical, which is used to reduce transp along lines of path types. In general, the sole purpose of hcomp is to allow terms to be

definitionally adjusted on a fragment of the type they live in. More precisely, suppose the ambient context Γ contains a number of path variables $\Phi = j, k, \ldots$ and suppose $\Gamma \vdash \chi : \mathsf{I}$. Intuitively, the term $\Gamma \vdash \mathsf{hcomp}\ \{B : \mathsf{Type}\}\ \{\chi : \mathsf{I}\}\ v\ (v_0 : B)$ is $v_0 : B$ but definitionally adjusted (using the v argument, not explained here) on the fragment of $\Gamma \vdash B$ where $\chi(j, k, \ldots) = \mathsf{i1}$ (since v_0, B, χ live in context Γ they can depend on variables in Φ). In cubical type theory, constraints like $\chi(j, k, \ldots) = \mathsf{i1}$ are called *face constraints* or alternatively *cofibrations* and can be regarded as subsets of a cube $\chi \subseteq \Phi$. The language used to express face constraints is called a *face or cofibration logic*. The cofibration logic used by Agda --cubical is De Morgan algebra and assertions in this logic are encoded as terms $\chi : \mathsf{I}$. If $\Gamma \vdash j, k, l : \mathsf{I}$, an example of face constraint is $\chi = ((\sim j) \vee k) \wedge l$.

In our case we would like, in a context extended with $(x : \mathsf{BI})$, to definitionally adjust the term $\mathsf{transp}\ (i.\, A\, i\, x)\ \varphi\ (u_0\ x)$ of type $A\,\mathsf{i1}\,x$ when $x = \mathsf{bi0}$ and $x = \mathsf{bi1}$ (and in fact when $\varphi = \mathsf{i1}$). The problem of course is that x is a bridge variable, and that hcomp only allows constraints $\varphi : I$ on path variables. The *mixed homogeneous composition* mhcomp primitive of Agda --bridges generalizes hcomp w.r.t. bridge variables and can be used in place of hcomp to specify the result of transporting along a line of bridges.

5.2 Mixed Homogeneous Composition

The mhcomp primitive of Agda --bridges has a type different than that of hcomp.

$$\mathsf{mhcomp} : \forall\ \{A : \mathsf{Type}\}\ \{\zeta : \mathsf{MCstr}\}\ (u : (i : \mathsf{I}) \to \mathsf{MPartial}\ \zeta\ A)\ (u_0 : A) \to A$$

This time u_0 and its type A can have free path variables $\Phi = (j, k, \ldots)$ but also free bridge variables $\Psi = (x, y, \ldots)$ and u_0 ought to be definitionally adjusted on a subset ζ of the mixed cube $\Phi \times \Psi$. For this reason mhcomp expects face constraints ζ expressed in an extended cofibration logic called MCstr. Concretely, the latter is a type postulated by Agda --bridges and equipped with primitives for combining atomic mixed face constraints. Instead of precisely explaining these primitives we provide a formula $\mathsf{MCstr}(\Phi, \Psi)$ expressing what mixed constraints ζ can be built in a context containing Φ and Ψ as above. First, define $\mathsf{I}(\Phi) = \{\varphi \mid \Phi \vdash \varphi : \mathsf{I}\}$. Second, define the set of bridge hyperfaces of Ψ as $H(\Psi) = \Psi \times \{\mathsf{bi0}, \mathsf{bi1}\}$. We define $\mathsf{BCstr}(\Psi) = \left\{ \bigvee_{(x,\mathsf{bi}\varepsilon) \in H'} (x = \mathsf{bi}\varepsilon) \,\middle|\, H' \subseteq H \right\} \cup \{\top\}$, i.e., bridge face constraints obtainable in Ψ are disjunctions of bridge hyperfaces (this includes an empty disjunction $\bot$), or a vacuous constraint denoted $\top$. Finally we set

$$\mathsf{MCstr}(\Phi, \Psi) = \frac{\mathsf{I}(\Phi) \times \mathsf{Bcstr}(\Psi)}{\forall \varphi\ \psi.\ (\mathsf{i1}, \psi) = (\varphi, \top) =: \top_{\mathsf{MCstr}}}$$

The quotient is taken to turn the map $\varphi \mapsto (\varphi, \bot)$ into an *embedding* of logics: a --cubical constraint $\varphi : \mathsf{I}$ holds if and only if its image $(\varphi, \bot) : \mathsf{MCstr}$ is a mixed constraint that holds. This condition is required to ensure that a term typechecks in Agda --cubical if and only if it typechecks in Agda --bridges. An example of mixed constrainted $\zeta : \mathsf{MCstr}$ is $\zeta := (\varphi, (x = \mathsf{bi0}) \vee (x = \mathsf{bi1}))$ which appears when transporting bridges, as hinted above.

$$\begin{aligned}&\mathsf{transp}\ (i.\, \mathsf{BridgeP}_{x.\,A\,i\,x}\ (a_0\ i)\ (a_1 i))\ \varphi\ u_0 \ \mapsto \\ &\lambda(x : \mathsf{BI}).\ \mathsf{mhcomp}\ \{A\,\mathsf{i1}\,x\}\{(\varphi, (x = \mathsf{bi0}) \vee (x = \mathsf{bi1}))\}\ (\ldots)\ (\mathsf{transp}\ (i.\, A\, i\, x)\ \varphi\ (u_0\ x))\end{aligned}$$

Similar to transp, the operational semantics of mhcomp is defined by induction on the syntax of its $A : \mathsf{Type}$ argument. Concretely, Agda --bridges duplicates the hcomp equations for Glue, hcomp, PathP, Σ, Π, record and (non-indexed, non-HIT) data types, but propagating a mixed constraint ζ this time. Additionally, it implements reductions at BridgeP and Gel types. The latter clause uses capturing. Following --cubical, if A is a HIT, an inhabitant $\mathsf{mhcomp}\ \{A\}\ \{\zeta\}\ u\ u_0 : A$ is considered normal and functions defined by pattern matching on A compute on it if $\zeta = (\varphi, \bot)$ for some φ.

Comparison with the CH Theory. While the theory of Agda --bridges extends CCHM cubical type theory, the CH theory is an extension of cartesian cubical type theory [Angiuli et al. 2021a, 2018] (CCTT). This distinction leads to Kan operations formulated differently in each system. First, in order to validate univalence, CCTT postulates V-types whereas CCHM postulates Glue types. Hence transp and mhcomp both have a reduction clause for Glue types instead of V-types. Second, the composition Kan operation of CCTT uses a simple cofibration logic with a constructor for diagonal constraints ($x = y : \mathbb{I}$). By contrast, CCHM uses a De Morgan cofibration logic which Agda --bridges had to extend (see MCstr above). To pinpoint a sound definition for MCstr we inspected the presheaf model $\mathsf{psh}(\square_{\mathsf{DM}} \times \square_{\mathsf{a}})$ where $\square_{\mathsf{DM}}$ (resp. $\square_{\mathsf{a}}$) is the category of De Morgan cubes (see CCHM; resp. affine cubes, see CH). The above formula for $\mathsf{MCstr}(\Phi, \Psi)$ can be regarded as the definition of such a presheaf. Finally some equations for the Kan operations use capturing, handled differently in Agda --bridges and CH: sound capturing is performed through context restriction in the CH theory (see Section 2.2) and through semi-freshness in Agda --bridges (see Section 2.4).

6 RELATED WORK

6.1 Relational Parametricity in and of Type Theory

In order to discuss and classify related work about parametricity in and of type theory, we analyze the general statement of parametricity. We assume that we are studying an object language $\mathscr{X}$ which embeds or can be interpreted in a target language $\mathscr{Y}$ where free theorems [Wadler 1989] will be stated.

> **Parametricity:** For every ($\mathscr{S}$) type T in $\mathscr{X}$, there exists a logical relation $[T]$ in $\mathscr{Y}$, such that every ($\mathscr{P}$) program $t : T$ in $\mathscr{X}$ is self-related according to $[T]$ in $\mathscr{Y}$.

Note that the general statement of parametricity contains two universal quantifications, which we have labeled with names $\mathscr{S}$ and $\mathscr{P}$ for the (meta)theories where these quantifications take place. We will classify related work on parametricity by its choice of languages/theories $\mathscr{X}$, $\mathscr{Y}$, $\mathscr{P}$ and $\mathscr{S}$. Note that if $\mathscr{X} = \mathscr{Y} = \mathscr{P}$, then we have global free theorems in $\mathscr{Y}$ and can prove in $\mathscr{Y}$ e.g. that Church encodings in $\mathscr{X}$ behave as the data type they encode. By contrast if $\mathscr{P}$ is a metatheory, then we only have free theorems for concretely known programs and the framework is merely a parametricity *translation* of such programs. These situations are often referred to as *internal* vs. *external* parametricity. In any provenly sound framework for internal parametricity hitherto developed that we are aware of, $\mathscr{S}$ is a metatheory (at least if you want $[T]$ to be a concrete relation and not a bridge type whose meaning needs to be characterized separately), implying that the SRP (Section 3.1.2) is a metatheorem.

Languages with internal parametricity typically feature non-standard parametricity primitives which call for a denotational model to establish soundness. In this case, when relevant, we specify how (closed) types in $\mathscr{Y}$ are modelled, and the metatheory $\mathscr{M}$ in which the model is built. In the case of external parametricity, we rather specify how $\mathscr{X}$ is interpreted in the metatheory $\mathscr{Y}$.

The resulting classification is given in Table 1 (Ab denotes Agda --bridges). The first block lists treatments of System F. Reynolds's original model is in set theory, but was later shown not to support impredicativity [Reynolds 1984]. Subsequent treatments use a logic over System F and prove parametricity results about concrete programs of concrete types. Atkey explains how Reynolds's model can be restructured as a reflexive graph model and then generalizes to System Fω. The third block lists treatments of external parametricity for dependent types. The first three papers prove parametricity results again in dependent type systems, but only for concrete programs of concrete types. The latter two use denotational models. Tabareau et al.'s approach is peculiar in that it defines the logical relation on the universe as the type of relations *that are the graph of an equivalence*, thus establishing a framework not for relational but for univalent parametricity.

Table 1. Classification of related work about parametricity, based on [Nuyts et al. 2017, fig. 9]. Here, $\mathscr{Y}$ is (faintly) highlighted when (possibly) $\mathscr{Y} = \mathscr{X}$; $\mathscr{P}$ is highlighted if $\mathscr{P} = \mathscr{Y} = \mathscr{X}$; and $\mathscr{S}$ is highlighted if $\mathscr{S} = \mathscr{X}$.

Citation	Obj. lang. $\mathscr{X}$	Target lang. $\mathscr{Y}$	$\mathscr{P}$	$\mathscr{S}$	$\mathscr{M}$	model in $\mathscr{Y}$ or *(italic)* $\mathscr{M}$
[Reynolds 1983]	System F	Meta: Set theory	$\mathscr{Y}$	$\mathscr{Y}$		Sets with relations
[Abadi et al. 1993]	System F	System $\mathscr{R}$	Meta	Meta	Meta	*PERs [Bellucci et al. 1995]*
[Plotkin and Abadi 1993]	System F	System F + logic	Meta	Meta		
[Wadler 2007]	System F	System F + logic	Meta	Meta		
[Atkey 2012]	System Fω	Meta: Impred. CIC	$\mathscr{Y}$	$\mathscr{Y}$		Reflexive graphs
[Takeuti 2001]	$\mathscr{X} \in \lambda$-cube	$\mathscr{X}$+$\forall$+$\Pi \in \lambda$-cube	Meta	Meta		
[Bernardy et al. 2012]	Any PTS	Suitable PTS	Meta	Meta		
[Tabareau et al. 2021]	CIC	Univalent CIC	Meta	Meta		
[Krishnaswami and Dreyer 2013]	CC	Meta	Meta	Meta		PERs
[Atkey et al. 2014]	MLTT	Meta: CIC	$\mathscr{Y}$	$\mathscr{Y}$		Reflexive graphs
[Bernardy and Moulin 2012]	BM	$\mathscr{X}$	$\mathscr{X}$	$\mathscr{X}$		*none*
[Bernardy et al. 2015]	BCM	$\mathscr{X}$	$\mathscr{X}$	Meta	Meta	*Affine cubical sets*
[Nuyts et al. 2017]	ParamDTT	$\mathscr{X}$	$\mathscr{X}$	Meta	Meta	*Bridge/path cubical sets*
[Nuyts and Devriese 2018]	RelDTT	$\mathscr{X}$	$\mathscr{X}$	Meta	Meta	*Depth n cubical sets*
[Cavallo and Harper 2021]	CH	$\mathscr{X}$	$\mathscr{X}$	Meta	Meta	*Affine/cart. bicub. sets*
Agda --bridges (Ab)	Ab	$\mathscr{X}$	$\mathscr{X}$	Meta	Meta	*Affine/CCHM bicub. sets*
ROTT	DTT ($\rightsquigarrow$ Ab)	Ab	$\mathscr{Y}$	$\mathscr{Y}$		RRGs
Corollary 4.1	Pr. Sys. F	Ab	$\mathscr{Y}$	$\mathscr{Y}$		RRGs
[Altenkirch et al. 2024]	ACKS	$\mathscr{X}$	$\mathscr{X}$	$\mathscr{X}$	Meta	*Affine cubical sets*
Cub. ROTT (envisioned)	$\approx$ ACKS	$\mathscr{X}$	$\mathscr{X}$	$\mathscr{X}$	Ab	*Relativistic cubical sets*

The fourth block lists treatments of internal parametricity for dependent types; these produce concrete free theorems (i.e. not mentioning bridge types that need to be characterized separately) for abstract programs of concrete types. Bernardy and Moulin's [2012] system predates the usage of named relational dimensions, but it is observational, i.e. the SRP holds definitionally. We are unaware of any soundness proof for this system. Bernardy et al. introduce the bridge interval as well as the extent and Gel combinators, and Cavallo and Harper combine their system with HoTT and demonstrate its power on paper. With Agda --bridges, we provide an implementation. The work on ParamDTT and RelDTT introduces a modal system in order to prove Reynolds's identity extension lemma for large types, but in the process has to adopt a cartesian cubical model which does not validate extent. As a consequence, these systems lack the power to prove parametricity of System F, which we demonstrated can be done in Agda --bridges (Section 4.3). We refer to Nuyts [2021] for a brief discussion of various internal parametricity features and their requirements in the model.

Strictly speaking, ROTT is again a dependently typed system with external parametricity that could go in the third block. However, ROTT was conceived as a commodity for Agda --bridges and seeks to obtain free theorems there. This is in contrast with e.g. Atkey et al. [2014], where MLTT is the system of interest but free theorems are obtained in some metatheory. We remark that since param is an external rule, the source syntax of ROTT is really just dependent type theory (extensible with bridge types, which are also displayed RRGs).

The reason why ROTT cannot provide internal parametricity is that the logical relation specified for an RRG, is itself not a displayed RRG (i.e. a dependent ROTT type) but only an external Agda --bridges type. This might be remedied in the future by moving from RRGs to relativistic cubical types, i.e. cubical types whose n-cubes are equivalent to n-cubes of bridges. Such a system would allow internal parametricity and satisfy the SRP definitionally. The syntax of such a system would

be very close to that of the concurrent work by Altenkirch et al. [2024], who present an internally parametric type theory which avoids the use of an interval and validates the SRP definitionally.

6.2 Parametricity and Univalence

While relational parametricity requires functions to respect relations, HoTT asks that they respect equivalences. Of course, equivalences are a form of relations, so one idea is akin to the other.

We already mentioned Tabareau et al.'s [2021] work in the previous section. A reformulation of their work using our techniques would effectively lead to a shallow embedding of observational setoid [Pujet and Tabareau 2022] or homotopy [Altenkirch et al. 2022] type theory in a CwF of univalent setoids or groupoids.

Awodey et al. [2018] define Church-like encodings of ordinary but also higher inductive types in (homotopy) type theory. Since the correctness of the Church encoding relies on preservation not only of equivalences but of all relations, they cannot be proven correct in plain type theory or HoTT. Instead, the authors enforce relational parametricity simply by adding it as a "such that" clause to the encoding.

6.3 The Structure Identity Principle (SIP)

We discuss appearances of the SIP in the literature and compare these to our discussion in Section 3. The HoTT book does not actually feature the full SIP and two other treatments rely on a DSL.

Standard notions of structure on univalent categories. The HoTT book [Program 2013] defines a *notion of structure* on a category C essentially as a displayed category $\mathcal{D}^\bullet$ over C such that the projection functor $P : \Sigma C \mathcal{D}^\bullet \to C$ from the total space is faithful, implying that the fiber of any object of C (and its identity morphism) is a preorder. A notion of structure is *standard* if this preorder is always a partial order, which is defined as a univalent preorder. The theorem called SIP then states that if C is univalent and $\mathcal{D}^\bullet$ is standard, then the total space $\Sigma C \mathcal{D}^\bullet$ is univalent. Relevant examples are group structures over h-sets, setoid structures over h-sets, monad structures over endofunctors, functor structures over indexed objects, …

Fundamentally, the proof of this theorem does two things: It applies the extensionality principle for Σ-types to characterize a path between objects in $\Sigma C \mathcal{D}^\bullet$, and it uses path induction to deduce "displayed" univalence from standardness (which meant fiberwise univalence). Importantly, if $\mathcal{D}^\bullet$ is quite complex, it is still up to the user to prove fiberwise univalence there, which still requires either rote work or the usage of a DSL for standard notions of structure. As such, we see this SIP as only a fragment of the fully general SIP.

A DSL for univalent structures on Type. Angiuli et al. [2021b] are concerned with proving the SIP (in our most general sense) for types of the form $T = \Sigma[\, X : \mathsf{Type} \,]\, \Sigma[\, s : S\, X \,]\, P\, X\, s$, where $P\, X\, s$ is a mere proposition (h-prop). The idea is that a tuple (X, s, p) is an algebra-like object with carrier X and operations s satisfying the axioms $P\, X\, s$. Their paper features a theorem titled SIP which amounts to the characterization of paths in a type of the form $\Sigma[\, X : \mathsf{Type} \,]\, C\, X$. Since paths in a mere proposition can be characterized as informationless (Theorem 3.6), only the SIP for the operations type $S\, X$ remains to be dealt with. A DSL – essentially the type language of the STLC with base type X (the carrier) – is then provided to build structures satisfying the SIP.

A DSL for univalent higher categories. Ahrens et al. [2020, thm. 7.10] use FOLDS [Makkai 1995] as a DSL for building univalent higher categories. In other words, they show that a higher type satisfies the SIP if it occurs as the object type of a higher category specified by a FOLDS signature.

ACKNOWLEDGMENTS

We thank Andrea Vezzosi for continuously sharing with us his expertise and sound suggestions regarding Agda --cubical and Agda --bridges. We thank Rasmus Møgelberg and Andrea Vezzosi for welcoming the first author at the IT University of Copenhagen. We thank the reviewers for their remarks and suggestions. Antoine Van Muylder holds a PhD fellowship (11H9921N) of the Research Foundation – Flanders (FWO). Andreas Nuyts holds a Postdoctoral fellowship (1247922N) of the Research Foundation – Flanders (FWO). This research is partially funded by the Research Fund KU Leuven and by the Research Foundation - Flanders (FWO; G030320N).

DATA AVAILABILITY STATEMENT

We provide in [Van Muylder et al. 2023] a virtual machine where Agda --bridges is installed and where our Agda --bridges library and its results are typechecked. The Agda --bridges implementation is developed in https://github.com/antoinevanmuylder/agda/tree/bridges and our library can be found in https://github.com/antoinevanmuylder/bridgy-lib. For some documentation, see the latter repository. Note that some notions have slightly different names than in this work.

REFERENCES

Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. 1993. Formal Parametric Polymorphism. *Theor. Comput. Sci.* 121, 1&2 (1993), 9–58. https://doi.org/10.1016/0304-3975(93)90082-5

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/J.TCS.2005.06.002

The Agda Community. [n. d.]. A standard library for Cubical Agda. https://github.com/agda/cubical

Agda Development Team. 2023. Agda 2.6.3 documentation. https://agda.readthedocs.io/en/v2.6.3/

Benedikt Ahrens and Peter LeFanu Lumsdaine. 2019. Displayed Categories. *Log. Methods Comput. Sci.* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:20)2019

Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. 2020. A Higher Structure Identity Principle. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM, 53–66. https://doi.org/10.1145/3373718.3394755

Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. 2024. Internal parametricity, without an interval. In *To appear in: Proceedings of the 51st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2024*. ACM.

Thorsten Altenkirch and Ambrus Kaposi. 2015. Towards a Cubical Type Theory without an Interval. In *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia (LIPIcs, Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:27. https://doi.org/10.4230/LIPICS.TYPES.2015.3

Thorsten Altenkirch, Ambrus Kaposi, and Michael Shulman. 2022. Towards Higher Observational Type Theory. In *28th International Conference on Types for Proofs and Programs (TYPES 2022). University of Nantes.*

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now!. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, Aaron Stump and Hongwei Xi (Eds.). ACM, 57–68. https://doi.org/10.1145/1292597.1292608

Abhishek Anand and Greg Morrisett. 2017. Revisiting Parametricity: Inductives and Uniformity of Propositions. *CoRR* abs/1705.01163 (2017). arXiv:1705.01163 http://arxiv.org/abs/1705.01163

Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. 2021a. Syntax and models of Cartesian cubical type theory. *Math. Struct. Comput. Sci.* 31, 4 (2021), 424–468. https://doi.org/10.1017/S0960129521000347

Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021b. Internalizing representation independence with univalence. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434293

Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. 2018. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK (LIPIcs, Vol. 119)*, Dan R. Ghica and Achim Jung (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:17. https://doi.org/10.4230/LIPICS.CSL.2018.6

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 46–61. https://doi.org/

10.4230/LIPICS.CSL.2012.46

Robert Atkey, Neil Ghani, and Patricia Johann. 2014. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 503–516. https://doi.org/10.1145/2535838.2535852

Steve Awodey, Jonas Frey, and Sam Speight. 2018. Impredicative Encodings of (Higher) Inductive Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 76–85. https://doi.org/10.1145/3209108.3209130

Roberto Bellucci, Martín Abadi, and Pierre-Louis Curien. 1995. A Model for Formal Parametric Polymorphism: A PER Interpretation for System R. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 902)*, Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin (Eds.). Springer, 32–46. https://doi.org/10.1007/BFB0014043

Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*, Dan R. Ghica (Ed.). Elsevier, 67–82. https://doi.org/10.1016/J.ENTCS.2015.12.006

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free - Parametricity for dependent types. *J. Funct. Program.* 22, 2 (2012), 107–152. https://doi.org/10.1017/S0956796812000056

Jean-Philippe Bernardy and Guilhem Moulin. 2012. A Computational Interpretation of Parametricity. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 135–144. https://doi.org/10.1109/LICS.2012.25

Auke Bart Booij, Martín Hötzel Escardó, Peter LeFanu Lumsdaine, and Michael Shulman. 2016. Parametricity, Automorphisms of the Universe, and Excluded Middle. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016, May 23-26, 2016, Novi Sad, Serbia (LIPIcs, Vol. 97)*, Silvia Ghilezan, Herman Geuvers, and Jelena Ivetic (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:14. https://doi.org/10.4230/LIPICS.TYPES.2016.7

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 182–194. https://doi.org/10.1145/3018610.3018620

Evan Cavallo. 2020. *Ptt, an experimental implementation of Martin-Löf type theory with n-ary internal parametricity.* https://github.com/ecavallo/ptt

Evan Cavallo and Robert Harper. 2019. Parametric Cubical Type Theory. *CoRR* abs/1901.00489 (2019). arXiv:1901.00489 http://arxiv.org/abs/1901.00489

Evan Cavallo and Robert Harper. 2021. Internal Parametricity for Cubical Type Theory. *Log. Methods Comput. Sci.* 17, 4 (2021). https://doi.org/10.46298/LMCS-17(4:5)2021

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2017. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *FLAP* 4, 10 (2017), 3127–3170. http://collegepublications.co.uk/ifcolog/?00019

Jean-Yves Girard. 1986. The System F of Variable Types, Fifteen Years Later. *Theor. Comput. Sci.* 45, 2 (1986), 159–192. https://doi.org/10.1016/0304-3975(86)90044-7

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Éditeur inconnu.

Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. 2013. Logical Relations and Parametricity - A Reynolds Programme for Category Theory and Programming Languages. In *Proceedings of the Workshop on Algebra, Coalgebra and Topology, WACT 2013, Bath, UK, March 1, 2013 (Electronic Notes in Theoretical Computer Science, Vol. 303)*, John Power and Cai Wingfield (Eds.). Elsevier, 149–180. https://doi.org/10.1016/J.ENTCS.2014.02.008

Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 381–395. https://doi.org/10.4230/LIPICS.CSL.2012.381

Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy (LIPIcs, Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 432–451. https://doi.org/10.4230/LIPICS.CSL.2013.432

Daniel Leivant. 1991. Finitely Stratified Polymorphism. *Inf. Comput.* 93, 1 (1991), 93–113. https://doi.org/10.1016/0890-5401(91)90053-5

Michael Makkai. 1995. First order logic with dependent sorts, with applications to category theory. (1995). http://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf

Bassel Mannaa and Rasmus Ejlers Møgelberg. 2018. The Clocks They Are Adjunctions Denotational Semantics for Clocked Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs, Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:17. https://doi.org/10.4230/LIPICS.FSCD.2018.23

Guilhem Moulin. 2016. *Internalizing Parametricity.* Ph. D. Dissertation. Chalmers University of Technology, Gothenburg, Sweden. http://publications.lib.chalmers.se/publication/235758-internalizing-parametricity

Andreas Nuyts. 2021. Parametricity Features and their Requirements. *CoRR* abs/2111.09822 (2021). arXiv:2111.09822 https://arxiv.org/abs/2111.09822

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 779–788. https://doi.org/10.1145/3209108.3209119

Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.* 1, ICFP (2017), 32:1–32:29. https://doi.org/10.1145/3110276

Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 664)*, Marc Bezem and Jan Friso Groote (Eds.). Springer, 361–375. https://doi.org/10.1007/BFB0037118

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study. https://homotopytypetheory.org/book/

Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. https://doi.org/10.1145/3498693

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard J. Robinet (Ed.). Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

John C. Reynolds. 1984. Polymorphism is not Set-Theoretic. In *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings (Lecture Notes in Computer Science, Vol. 173)*, Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin (Eds.). Springer, 145–156. https://doi.org/10.1007/3-540-13346-1_7

Egbert Rijke. 2022. Introduction to Homotopy Type Theory. arXiv:2212.11082 [math.LO]

Robert Rose, Matthew Z Weaver, and Daniel R Licata. 2022. Deciding the cofibration logic of cartesian cubical type theories. In *28th International Conference on Types for Proofs and Programs (TYPES 2022). University of Nantes.*

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The Marriage of Univalence and Parametricity. *J. ACM* 68, 1, Article 5 (jan 2021), 44 pages. https://doi.org/10.1145/3429979

Izumi Takeuti. 2001. The Theory of Parametricity in Lambda Cube. Technical report 1217, Kyoto University. https://repository.kulib.kyoto-u.ac.jp/dspace/bitstream/2433/41237/1/1217_10.pdf

The Coq development team. 2022. The Coq proof assistant. http://coq.inria.fr

Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. 2023. *Agda --bridges VM.* https://doi.org/10.5281/zenodo.10009365

Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-calculus in guarded cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 270–283. https://doi.org/10.1145/3372885.3373814

Niccolò Veltri and Andrea Vezzosi. 2023. Formalizing CCS and π-calculus in Guarded Cubical Agda. *J. Log. Algebraic Methods Program.* 131 (2023), 100846. https://doi.org/10.1016/J.JLAMP.2022.100846

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.* 31 (2021), e8. https://doi.org/10.1017/S0956796821000034

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler. 2007. The Girard-Reynolds isomorphism (second edition). *Theor. Comput. Sci.* 375, 1-3 (2007), 201–226. https://doi.org/10.1016/J.TCS.2006.12.042

Received 2023-07-11; accepted 2023-11-07