KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Parametricity in Type Theory: Language Primitives and Applications

Antoine Van Muylder

Dissertation presented in partial fulfillment
of the requirements for the degree of
Doctor of Engineering Science (PhD):
Computer Science

Supervisors:
Prof. dr. Dominique Devriese
Dr. Andreas Nuyts

April 2026

PARAMETRICITY IN TYPE THEORY: LANGUAGE PRIMITIVES AND APPLICATIONS

Antoine VAN MUYLDER

Supervisors:
Prof. dr. Dominique Devriese
Dr. Andreas Nuyts

Members of the
Examination Committee:
Prof. dr. Lucie Vandewalle, chair
Prof. dr. Yolande Berbers
Prof. dr. Franck Piessens
Prof. dr. Tom Schrijvers
Prof. dr. Andrew M. Pitts
(University of Cambridge)
Dr. Robert Atkey
(University of Strathclyde)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

April 2026

Uitgegeven in eigen beheer, Antoine Van Muylder, Celestijnenlaan 200A box 2402 (Belgium)

Acknowledgments

I am deeply grateful to my supervisors Dominique Devriese and Andreas Nuyts for their genuine support, their expertise, and critical feedback throughout the years. I especially thank Dominique for encouraging me to work on Kripke parametricity and Andreas for sharing his knowledge about the syntax and semantics of languages.

I also want to thank Andrea Vezzosi and Rasmus Møgelberg for welcoming me to the IT University of Copenhagen during the early stages of this project. I am particularly grateful to Andrea for introducing me to the inner workings of Agda --cubical and for sharing his proof of the adequacy of PHOAS.

I thank the Research Foundation – Flanders (FWO) for the doctoral fellowship (FWO; 11H9921N) that enabled me to conduct my doctoral research over four years. The research was subsequently also funded by the Research Fund KU Leuven and the Internal Funds KU Leuven. Additionally, Andreas Nuyts holds a Postdoctoral Fellowship from the Research Foundation - Flanders (FWO; 12AB225N).

I thank each member of my PhD jury for their time and helpful comments.

I thank my former colleagues and advisors Cătălin Hriţcu, Exequiel Rivas, and Kenji Maillard for welcoming me as an intern before this doctoral project, and for everything I learned from them about programming languages and mathematics.

Lastly, I thank my parents and sister for their incredible support, and my colleagues and office mates for making the office a friendly place to work.

Popularized Abstract

Software errors can have serious consequences. Formal verification is an approach that addresses this by replacing testing with mathematical proof: instead of running a program and checking whether it behaves correctly on a sample of inputs, one constructs a rigorous argument that it behaves correctly on all inputs. This proof can itself be checked by a computer, eliminating the risk of human oversight. Tools that support this kind of work are called proof assistants.

This thesis contributes to the science of making proof assistants more powerful and easier to use. The specific topic is parametricity, a principle concerning generic programs, i.e. programs expecting a type as an input. An example of a generic program is reverseList : $(A : \mathsf{Type}) \to \mathsf{List}\ A \to \mathsf{List}\ A$, which expects two inputs (a type A and a list of elements of A) and outputs the input list in reverse order. Observe that the reverseList program does not need to inspect what exact type A is to operate, its behavior is similar regardless of its type input. Parametricity is an axiom asserting that all generic programs have this property: they do not use their type input in their implementation. If parametricity holds, from the mere type signature of a generic program (its input and output types) one can derive non-trivial mathematical theorems about it, without even reading the program's body. These are called theorems for free, and they have been central to programming language theory since the 1980s.

The first contribution of this thesis is Agda --bridges, the first full proof assistant making parametricity an intrinsic feature of the language. Built on top of the existing Agda proof assistant, it allows parametricity-based theorems to be verified directly on a computer, rather than only on paper. Alongside it, a library called ROTT reduces the practical effort of deriving theorems for free to a one-liner, rather than a laborious manual exercise.

The second contribution connects parametricity to a different problem: how programming languages handle names. Programs routinely manipulate names: variable names in a compiler, identifiers in a database, labels in a network protocol. Reasoning about names is surprisingly subtle: when are two expressions "the same up to renaming"? What does it mean for a name to be "fresh"? This thesis shows that these questions have clean answers when viewed through the lens of a nullary

variant of parametricity. The result is a new formal system, Parametric Nominal Type Theory, which handles names in a principled and mathematically coherent way.

Gepopulariseerde Samenvatting

Softwarefouten kunnen ernstige gevolgen hebben. Formele verificatie pakt dit probleem aan door testen te vervangen door wiskundige bewijzen: in plaats van een programma uit te voeren op een steekproef van invoeren en te controleren of het correct werkt, construeert men een rigoureus argument dat het correct werkt op alle invoeren. Dit bewijs kan door een computer worden gecontroleerd, waardoor het risico op menselijk fouten bij de controle wordt weggenomen. Hulpmiddelen die dit soort werk ondersteunen, worden bewijsassistenten genoemd.

Dit proefschrift draagt bij aan de wetenschap van het krachtiger en gebruiksvriendelijker maken van bewijsassistenten. Het specifieke onderwerp is parametriciteit, een principe dat betrekking heeft op generieke programma's, d.w.z. programma's die een type als invoer verwachten. Een voorbeeld van een generiek programma is $\mathsf{reverseList} : (A : \mathsf{Type}) \to \mathsf{List}\, A \to \mathsf{List}\, A$, dat twee invoeren verwacht (een type A en een lijst van elementen van A) en de invoerlijst in omgekeerde volgorde teruggeeft. Merk op dat het programma reverseList niet hoeft te inspecteren wat het exacte type A is om te werken; zijn gedrag is gelijkaardig ongeacht zijn type-invoer. Parametriciteit is een axioma dat stelt dat alle generieke programma's deze eigenschap hebben: ze maken geen gebruik van hun type-invoer in hun implementatie. Als parametriciteit geldt, kan men louter uit de type-signatuur van een generiek programma (zijn invoer- en uitvoertypes) niet-triviale wiskundige stellingen over dat programma afleiden, zonder zelfs maar de implementatie te lezen. Deze worden "theorems for free" genoemd, en ze staan al sinds de jaren tachtig centraal in de theorie van programmeertalen.

De eerste bijdrage van dit proefschrift is Agda --bridges, de eerste volledige bewijsassistent die parametriciteit tot een intrinsiek kenmerk van de taal maakt. Voortgebouwd op de bestaande Agda-bewijsassistent, maakt Agda --bridges het mogelijk om op parametriciteit gebaseerde stellingen rechtstreeks op een computer te verifiëren, in plaats van alleen op papier. Onze bijhorende bibliotheek genaamd ROTT reduceert de praktische inspanning om theorems for free af te leiden tot één regel code, in plaats van een arbeidsintensieve handmatige oefening.

De tweede bijdrage verbindt parametriciteit met een ander probleem: hoe programmeertalen omgaan met namen. Programma's manipuleren routinematig namen: variabelenamen in een compiler, identificatoren in een databank, labels in een netwerkprotocol. Redeneren over namen is verrassend subtiel: wanneer zijn twee uitdrukkingen "hetzelfde op hernoeming na"? Wat betekent het dat een naam "vers" is? Dit proefschrift toont aan dat deze vragen heldere antwoorden hebben wanneer ze worden bekeken door de lens van een nulaire variant van parametriciteit. Het resultaat is een nieuw formeel systeem, Parametrische Nominale Typetheorie, dat namen op een principiële en wiskundig coherente manier behandelt.

Abstract

Formal software verification systems aim to provide rigorous mathematical proofs that programs adhere to their specifications. In particular, proof assistants like Agda, Rocq and Lean can express programs, specifications and proofs within a single unifying language called Dependent Type Theory (DTT).

This thesis makes contributions to parametricity within the setting of DTT and proof assistants. As a first approximation, parametricity is a uniformity property regarding polymorphic, i.e. generic programs. A generic program behaves identically regardless of the type it is instantiated with, because it cannot inspect its type argument. This simple observation leads to useful knowledge when performing proofs about the program (Wadler calls such knowledge "theorems for free").

More generally, Reynolds mathematically defined the notion of parametricity as relational parametricity, the statement that every type can be turned into a certain reflexive graph. An edge in the graph between two values is a proof that the values have a similar structure. For instance an edge between polymorphic programs is a proof that they map related types to related outputs, and this formalizes what it means to be uniform.

The parametricity translation, mapping types to reflexive graphs, is defined externally as a meta-operation on DTT expressions and is not an operation that is available inside, or internal to DTT. For example, given a type of polymorphic functions, one cannot prove inside DTT the formal statement expressing that every such function is uniform (we call such statements global free theorems).

Internally parametric type theories (PTTs) achieve this by equipping DTT with a so-called Bridge type former, whose role is to represent the parametricity translation inside the theory. A landmark example is the interval-based theory of Cavallo and Harper (the CH theory), in which bridges are functions from a postulated bridge interval, in analogy with the path types of cubical type theory. Within the CH theory, global free theorems become provable. However, a practical limitation persists: contrary to the Reynolds translation of a type, the Bridge translation of a type does not directly provide an actionable parametricity result. Indeed tedious case-by-case rote work is required of the user to establish that the Bridge type former commutes with each type former appearing in the type under consideration.

The first main contribution of this thesis is to improve the practical usability of internally parametric type theories. Firstly, we address the lack of a full-fledged proof-assistant implementation of binary internal parametricity and contribute the Agda --bridges proof assistant. Agda --bridges is an extension of the Agda proof assistant and implements an interactive typechecker for the CH parametric type theory. More precisely, Agda --bridges extends Agda --cubical, itself an implementation of cubical type theory. In fact, Agda --bridges typechecks the standard library of Agda --cubical, hence important theorems provided by Agda --cubical, like univalence, remain available to the user of Agda --bridges. Moreover, Agda --bridges validates key theorems for internal parametricity, in particular the relativity equivalence. This makes it possible to provide formal proofs of free theorems, including global ones. Yet the rote work challenge described above persists.

Hence, secondly, we contribute Relational Observational Type Theory (ROTT), a library, or domain-specific language, written in Agda --bridges that eliminates the rote work in a principled way. ROTT lets the user obtain concise and modular proofs of actionable parametricity statements. Once a type is written using the ROTT DSL, a corresponding proof can be extracted as a one-liner. Using this methodology we are able to formalize global free theorems of practical and theoretical relevance. Notably, we expand on a proof communicated to us by Andrea Vezzosi and can show that higher-order abstract syntax is an adequate representation of the untyped lambda calculus (previously known proofs relied on the strictly stronger notion of Kripke parametricity).

The second main contribution of this thesis concerns nullary internal parametricity and its relationship to the formal study of languages with variable binding. When studying a language on paper it is common to think of variables as strings, and to adopt the convention that α-equivalent terms are equal. However, when working formally it is preferable to represent the syntax of the object language in such a way that α-equivalent terms are equal by construction. Nominal frameworks are type systems featuring a so-called name abstraction type former, used to give a type to the binders of the object language. This type former enables a string-like but α-equivalence-respecting representation of syntax with binders. Yet existing nominal frameworks either feature typing rules hard to implement in a proof assistant environment, or lack expressivity to reason about nominal syntax.

We solve these issues by contributing Parametric Nominal Type Theory, which is an extension of the nullary CH theory, a version of the CH theory where bridges have 0 endpoints instead of 2. Firstly, we recognize that the nullary CH theory is itself the basis of a nominal framework in which name abstraction corresponds to the nullary bridge type former. In fact the other primitives of the CH theory can be understood from a nominal point of view and existing nominal primitives can be implemented in terms of the CH ones.

Secondly, we identify the missing piece that suffices to turn the nullary CH theory into an actual nominal framework, in which one can reason about object languages in

a nominal fashion without the aforementioned lack of expressivity. The missing piece is a type of names Nm such that its nullary Bridge type/nullary translation is the sum type $1 + \mathsf{Nm}$. We provide an induction principle for Nm that entails the property. We demonstrate that Parametric Nominal Type Theory is a suitable nominal framework. One of our examples involves emulating a restricted form of Kripke parametricity by nullary parametricity.

Beknopte samenvatting

Formele softwareverificatiesystemen hebben als doel rigoureuze wiskundige bewijzen te leveren dat programma's voldoen aan hun specificaties. In het bijzonder kunnen bewijsassistenten zoals Agda, Rocq en Lean programma's, specificaties en bewijzen uitdrukken binnen één verenigde taal die Afhankelijke Typetheorie (DTT) wordt genoemd.

Dit proefschrift levert bijdragen aan parametriciteit binnen de context van DTT en bewijsassistenten. Als eerste benadering is parametriciteit een uniformiteitseigenschap van polymorfische, i.e. generieke programma's: een generiek programma gedraagt zich identiek ongeacht het type waarmee het wordt geïnstantieerd, omdat het zijn type-argument niet kan inspecteren. Deze eenvoudige observatie leidt tot nuttige kennis bij het maken van bewijzen over dit programma (Wadler noemt dergelijke kennis "theorems for free").

Meer algemeen definieerde Reynolds het begrip parametriciteit wiskundig als relationele parametriciteit, de stelling dat elk type kan worden omgezet in een reflexieve graaf. Een boog in de graaf tussen twee waarden is een bewijs dat de waarden een gelijkaardige structuur hebben. Een boog tussen polymorfische programma's bijvoorbeeld is een bewijs dat ze gerelateerde types afbeelden op gerelateerde uitvoeren, en dit formaliseert wat het betekent om uniform te zijn.

De parametriciteitsvertaling, die types afbeeldt op reflexieve graven, is extern gedefinieerd als een meta-operatie op DTT-expressies en is geen operatie die beschikbaar is binnen, of intern aan DTT. Gegeven een type van polymorfische functies bijvoorbeeld, kan men binnen DTT niet bewijzen dat elke dergelijke functie uniform is (wij noemen zulke stellingen globale vrije theorema's).

Intern parametrische typetheorieën (PTT's) bereiken dit door DTT uit te breiden met een zogenoemde Bridge-typevormer, die de parametriciteitsvertaling binnen de theorie voorstelt. Een belangrijk voorbeeld is de intervalgebaseerde theorie van Cavallo en Harper (de CH-theorie), waarin bruggen functies zijn van een gepostuleerd bruginterval, naar analogie met de padtypes van kubische typetheorie. Binnen de CH-theorie worden globale vrije theorema's bewijsbaar. Er blijft echter een praktische beperking bestaan: in tegenstelling tot de Reynolds-vertaling van een type levert de Bridge-vertaling van een type geen direct bruikbaar parametriciteitsresultaat op.

Vervelend geval-per-geval routinewerk is namelijk vereist van de gebruiker om aan te tonen dat de Bridge-typevormer commuteert met elke typevormer die voorkomt in het beschouwde type.

De eerste hoofdbijdrage van dit proefschrift is het verbeteren van de praktische bruikbaarheid van intern parametrische typetheorieën. Ten eerste pakken we het gebrek aan een volwaardige bewijsassistentimplementatie van binaire interne parametriciteit aan en dragen we de Agda --bridges bewijsassistent bij. Agda --bridges is een uitbreiding van de Agda-bewijsassistent en implementeert een interactieve typechecker voor de CH parametrische typetheorie. Meer precies breidt Agda --bridges Agda --cubical uit, zelf een implementatie van kubische typetheorie. Agda --bridges typecheckt de standaardbibliotheek van Agda --cubical, waardoor belangrijke theorema's die door Agda -- worden geleverd, zoals univalentie, beschikbaar blijven voor de gebruiker van Agda --bridges. Bovendien valideert Agda --bridges sleuteltheorema's voor interne parametriciteit, in het bijzonder de *relativity equivalence*. Dit maakt het mogelijk formele bewijzen van vrije theorema's te leveren, inclusief globale. Het routinewerk-probleem dat we hierboven beschreven blijft echter bestaan.

Daarom dragen we, ten tweede, Relationele Observationele Typetheorie (ROTT) bij, een bibliotheek, of domeinspecifieke taal, geschreven in Agda --bridges die het routinewerk op een principiële manier elimineert. ROTT geeft de gebruiker beknopte en modulaire bewijzen van bruikbare parametriciteitsstellingen. Zodra een type is geschreven met behulp van de ROTT-DSL, kan een overeenkomstig bewijs worden geëxtraheerd als één regel code. Met deze methode zijn we in staat globale vrije theorema's met praktische en theoretische relevantie te formaliseren. Met name werkten we een bewijs verder uit dat ons werd meegedeeld door Andrea Vezzosi en kunnen we aantonen dat hogere-orde abstracte syntax een adequate representatie is van de ongetypeerde λ-calculus (eerder bekende bewijzen steunden op het strikt sterkere begrip van Kripke-parametriciteit).

De tweede hoofdbijdrage van dit proefschrift betreft nulaire interne parametriciteit en haar verhouding tot de formele studie van talen met variabelenbinding. Bij het bestuderen van een programmeertaal op papier is het gebruikelijk om variabelen als tekenreeksen te beschouwen en de conventie aan te nemen dat α-equivalente termen gelijk zijn. Bij formeel werk verdient het echter de voorkeur dat de syntax van de objecttaal zo wordt voorgesteld dat α-equivalente termen per constructie gelijk zijn. Nominale frameworks zijn typesystemen met een zogenoemde naamabstractie-typevormer, gebruikt om een type te geven aan de binders van de objecttaal. Deze typevormer maakt een string-achtige maar α-equivalentierespecterende representatie van syntax met binders mogelijk. Bestaande nominale frameworks hebben echter ofwel typeringsregels die moeilijk te implementeren zijn in een bewijsassistent, ofwel ontbreekt het hen aan expressiviteit om over nominale syntax te redeneren.

We lossen deze problemen op door Parametric Nominal Type Theory bij te dragen, een uitbreiding van de nulaire CH-theorie, een versie van de CH-theorie waarin

bruggen 0 eindpunten hebben in plaats van 2. Ten eerste erkennen we dat de nulaire CH-theorie zelf de basis vormt van een nominaal framework waarin naamabstractie overeenkomt met de nulaire Bridge-typevormer. De andere primitieven van de CH-theorie kunnen ook vanuit een nominaal standpunt worden begrepen en bestaande nominale primitieven kunnen worden geïmplementeerd in termen van de CH-primitieven.

Ten tweede identificeren we het ontbrekende stuk dat volstaat om de nulaire CH-theorie om te vormen tot een werkelijk nominaal framework, waarin men over objecttalen kan redeneren op een nominale manier zonder het eerder vermelde gebrek aan expressiviteit. Het ontbrekende stuk is een type voor namen Nm zodanig dat zijn nulaire Bridge-type/nulaire vertaling het somtype $1 + \mathsf{Nm}$ is. We leveren een inductieprincipe voor Nm dat deze eigenschap impliceert. We tonen aan dat Parametric Nominal Type Theory een geschikt nominaal framework is. Een van onze voorbeelden betreft het emuleren van een beperkte vorm van Kripke-parametriciteit door middel van nulaire parametriciteit.

Contents

Chapter 1

Introduction

Formal software verification is a branch of computer science dedicated to the development of techniques enabling rigorous mathematical proofs that programs adhere to their specification. To enable such rigor, both the program and its specification must be expressed in a formal language. Furthermore, in order to delegate to a computer the task of verifying that the proof contains no errors, the proof itself should be a formal expression as well.

A prominent class of modern formal verification systems, a.k.a *proof assistants* (e.g. Agda [Agd25], Rocq [Roc25], Lean [MU21]), adopts the perspective that programs, specifications and proofs should all be expressed in a single unifying language known as Dependent Type Theory (DTT; several kinds of DTTs exist and each proof assistant implements its own variation).

1.1 Proof Assistants and Dependent Types

In DTT, only two kinds of entities can be expressed: terms (i.e. programs) and types. Then specifications and proofs are particular cases of types and terms, respectively. Indeed any mathematical statement, such as the specification of a given program, can be faithfully encoded as a certain type P, because DTT offers a rich set of language primitives capable of expressing complex types. Furthermore proofs of the statement conveyed by the type P correspond precisely to terms $p : P$. In particular, verifying that p is a well-formed error-free proof of the statement P boils down to verifying whether p is a term of type P, i.e. whether $p : P$ typechecks. The latter correspondence between proofs of a mathematical statement on one side, and programs of a certain type on the other, is called the Curry-Howard correspondence. In short it abstracts away the notions of proof and statement, shifting the focus instead to terms, types, and typechecking.

1.1.1 Typechecking Proofs in Agda

We present a basic example to illustrate the proofs-as-programs paradigm. The example is written in the Agda [Agd25] programming language, the primary language of this thesis.

The Agda language is a particular DTT and as such can be classified as a *functional* programming language with the following characteristics: (1) the language is pure, e.g. has no primitives to mutate memory and (2) the language is total, meaning that programs that typecheck are guaranteed to terminate. In functional languages, common programs like swapHeadsNat below should be thought of as mathematical functions between types, and are defined by equations. As a point of reference, the syntax of Agda is close to the syntax of the better-known Haskell [Mar+10] functional programming language (in fact Agda is implemented in Haskell). On top of that Agda is also a proof assistant, or interactive theorem prover, in that the user of Agda can interact with Agda's typechecker in order to build correct proofs step by step, as shall be discussed below.

```
swapHeadsNat : List ℕ → List ℕ
swapHeadsNat [] = []
swapHeadsNat (n :: []) = (n :: [])
swapHeadsNat (hd0 :: hd1 :: tl) = hd1 :: hd0 :: tl

-- map : {A B : Type} → (A → B) → List A → List B
swapHeadsNat-map : (f : ℕ → ℕ) (ns : List ℕ) →
  map f (swapHeadsNat ns) ≡ swapHeadsNat (map f ns)
swapHeadsNat-map f [] = refl
swapHeadsNat-map f (n :: []) = refl
swapHeadsNat-map f (hd0 :: hd1 :: tl) = ?
```

The first block in the example defines a program swapHeadsNat, of type $\mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N}$. The swapHeadsNat function swaps the first two elements of its input list. More precisely, swapHeadsNat is defined by a case analysis on its input list. Case analyses are frequent in functional languages and several technical terms describe just that: so swapHeadsNat can alternatively be said to be defined (1) by pattern matching or (2) by recursion or (3) by elimination. The first two clauses in the definition of swapHeadsNat handle the cases where the input list of swapHeadsNat is either empty or of size 1. In these cases the input list is returned without any modification. The third clause of swapHeadsNat handles the case of an input list of the form $(\mathsf{hd}_0 :: \mathsf{hd}_1 :: \mathsf{tl})$, whose first two elements are $\mathsf{hd}_0, \mathsf{hd}_1$ and whose remaining elements are given by the list $\mathsf{tl} : \mathsf{List}\,\mathbb{N}$. In that case, swapHeadsNat returns the expected list $(\mathsf{hd}_1 :: \mathsf{hd}_0 :: \mathsf{tl})$.

The second block in the example relies on a certain function from Agda's standard library called map. As can be seen from its type written in a comment above, the map function expects 4 arguments: two types A, B, a function $f : A \to B$ and an input list $as : \mathsf{List}\,A$. Overall what the map operation does is output a list $\mathsf{map}\ f\ as$:

List B where each entry is the result of applying f on the corresponding entry in as. Note the following about the type of map. (1) In functional languages it is common and allowed to express the type of a function with several arguments by chaining together several functions types written with $\to$. By default the $\to$ symbol is right-associative, i.e. $X \to Y \to Z$ should be parsed as $X \to (Y \to Z)$. (2) map is dubbed a *dependent function* because its output type *depends* on, i.e. mentions as a free variable, its input A (this is also valid for the B input). (3) Curly braces { } may surround arguments of a function to mark them as implicit: conveniently such arguments must then not be supplied to the function when calling it, and are rather inferred by Agda from analyzing subsequent arguments. For instance the call map f typechecks since A, B are inferred from f's type $f : A \to B$. (4) In functional languages, a function call is rather dubbed a function application by reference to mathematical functions. Moreover, function application is denoted by a space between the callee and the argument and function application associates to the left by default. So the following composite function application map f (map g ms) should be parsed as (map f) ((map g) ms).

The second block in the example declares, and partially proves, a theorem called swapHeadsNat-map. Strictly speaking, in accordance with the Curry-Howard correspondence, the theorem is rather the *type* of the swapHeadsNat-map program, and the proof of the theorem is the swapHeadsNat-map program itself. This abuse of terminology is common in proof assistants. The swapHeadsNat-map theorem is a commutation result regarding the swapHeadsNat and map functions. Indeed the type of swapHeadsNat-map can be understood as a mathematical statement asserting that: for each function ($f : \mathbb{N} \to \mathbb{N}$) and list ($ns$: List $\mathbb{N}$), applying the swapHeadsNat and map f operations to ns in any order gives rise to equal lists in List $\mathbb{N}$. In other words, swapHeadsNat and map f commute with each other. As any other statement, the latter equality is expressed as a type. More precisely the equality statement is a specialization of Agda's equality type-former $\_\equiv\_ : \{L : \mathsf{Type}\} \to L \to L \to \mathsf{Type}$ where in particular L is set to List B (note that in Agda, underscores in names allow to define infix notations).

The swapHeadsNat program was defined by performing a case analysis on its input. Unsurprisingly the proof of swapHeadsNat-map proceeds by performing a similar case analysis on the input list ns. Again more technical terms exist to refer to such case analyses: the swapHeadsNat-map proof can alternatively be said to be defined or obtained (1) by dependent pattern matching (c.f. pattern matching in the case where the output type does not depend on the input ns) (2) by *induction* (c.f. recursion) (3) by dependent elimination (c.f. elimination). We explain how the proof proceeds for lists with ≥ 2 elements, the other cases are in fact similar. To finish the proof one has to replace the ? in the definition of swapHeadsNat-map with an actual term that typechecks. Conveniently, when Agda typechecks an expression with ?, it interactively provides to the user a certain "goal" displaying the expected type of ? and the hypotheses/arguments that are present in the context at that point. The goal

generated by Agda for the above ? looks like this:

```
-- Goal: map f (swapHeadsNat (hd0 :: hd1 :: tl)) ≡
--       swapHeadsNat (map f (hd0 :: hd1 :: tl))
-- ------------------------------------------
-- tl  : List ℕ
-- hd1 : ℕ
-- hd0 : ℕ
-- f   : ℕ → ℕ
```

From a user point of view this goal is easy to prove since ? can be replaced by the refl primitive, making the proof typecheck. From the point of view of the Agda typechecker, verifying that refl has the target equality type triggers a computation reducing both sides of the equality type according to computation rules known by the typechecker. Indeed, both swapHeadsNat and map f are defined by pattern matching and both functions compute on inputs of the form $(x :: y :: z)$ according to their defining pattern matching equations. That is to say, the following derivations can be computed by the Agda typechecker

$$\begin{aligned}&\mathsf{map}\ f\ (\mathsf{swapHeadsNat}\ (\mathsf{hd}_0 :: \mathsf{hd}_1 :: \mathsf{tl}))\\ &= \mathsf{map}\ f\ (\mathsf{hd}_1 :: \mathsf{hd}_0 :: \mathsf{tl})\\ &= ((f\ \mathsf{hd}_1) :: (f\ \mathsf{hd}_0) :: (\mathsf{map}\ f\ \mathsf{tl}))\end{aligned}$$

and

$$\begin{aligned}&\mathsf{swapHeadsNat}\ (\mathsf{map}\ f\ (\mathsf{hd}_0 :: \mathsf{hd}_1 :: \mathsf{tl}))\\ &= \mathsf{swapHeadsNat}\ ((f\ \mathsf{hd}_0) :: (f\ \mathsf{hd}_1) :: (\mathsf{map}\ f\ \mathsf{tl}))\\ &= ((f\ \mathsf{hd}_1) :: (f\ \mathsf{hd}_0) :: (\mathsf{map}\ f\ \mathsf{tl}))\end{aligned}$$

Thus the typechecker knows that the two sides of the target equality types are $=$ and refl typechecks. The equality $=$ involved in these derivations is different from the equality type $\equiv$ from above. The $=$ equality is called "definitional" and can be understood as a convertibility relation. Roughly speaking, two terms x, y are definitionally equal $x = y$ if unfolding definitions in both terms leads to syntactically equal terms (in reality, the Agda typechecker defines a precise set of rules governing definitional equality).

1.2 Parametricity, as a Uniformity Property

This thesis is centered around the notion of parametricity. Historically, parametricity was first understood as a property of polymorphic programs. The type of such programs p depends on a type variable, but their body does not, i.e. p is uniform. This simple fact leads to a series of interesting consequences regarding the behavior of p, known as theorems for free [Wad89].

1.2.1 Polymorphic Programs

Parametric polymorphism is a common feature of high-level programming languages. A parametrically polymorphic program (in short: a polymorphic or generic program) is simply a program p parameterized by a type variable A. For instance the swapHeadsNat program above may be generalized to the following polymorphic program swapHeads which works with all kinds of lists, not just lists of natural numbers.

```
swapHeads : (A : Type) → List A → List A
swapHeads A [] = []
swapHeads A (a :: []) = a :: []
swapHeads A (hd0 :: hd1 :: tl) = hd1 :: hd0 :: tl
```

Similar to other abstraction mechanisms in programming languages, the primary aim of parametric polymorphism is to allow the programmer to reduce code duplication and thus improve maintainability. Indeed with this mechanism the programmer may replace a potentially large set of similar-looking definitions of swapping functions $\mathsf{List}\, B \to \mathsf{List}\, B$ defined for concrete types B, with a single definition swapHeads : $(A : \mathsf{Type}) \to \mathsf{List}\, A \to \mathsf{List}\, A$.

So by design, parametrically polymorphic programs behave uniformly, that is, they operate in a similar fashion regardless of their type argument. Another way to say this is that their type argument is only used for typing purposes, not for computing purposes. For instance the dependent function swapHeads has an output type $\mathsf{List}\, A \to \mathsf{List}\, A$ that clearly depends on A but the A argument is not used in the definition of swapHeads. What's more, the primitives provided by Agda do not in fact provide any way to write a body for swapHeads A where A occurs. This would be possible if, for example, Agda offered a type-casing operation used to branch on concrete values of A, but Agda and other functional languages do not feature such an operation. Hence by mere virtue of having type $(A : \mathsf{Type}) \to \mathsf{List}\, A \to \mathsf{List}\, A$, the swapHeads program *is* parametric, i.e. behaves uniformly.

1.2.2 Theorems for Free

The fact that polymorphic programs p behave uniformly in their type argument may seem innocuous at first sight. Yet, this fact leads to a series of interesting logical consequences about p, called theorems for free by Wadler [Wad89], and sometimes just called free theorems. These can intuitively be explained as follows. Consider a parametrically polymorphic program $p : (A : \mathsf{Type}) \to \mathsf{List}\, A \to \mathsf{List}\, A$, whose implementation we do not know. Given an input type A and an input list $as : \mathsf{List}\, A$, the program p can only craft an output $p\, A\, as$ by using operations pertaining to lists, because p is parametric and thus cannot inspect A. In other words, the output $p\, A\, as : \mathsf{List}\, A$ will be a list like as but where some entries have been removed/duplicated/exchanged, and we deduced this property without looking at

p's implementation and from its type alone (and the fact that it is parametrically polymorphic). A formal consequence of this analysis regarding the output $p\,A\,as$ is that we expect the following equality to hold: $\mathsf{map}\,f\,(p\,A\,as) \equiv p\,B\,(\mathsf{map}\,f\,as)$ for any A, B types and $f : A \to B$. And indeed given a concrete, typechecked Agda program $p : (A : \mathsf{Type}) \to \mathsf{List}\,A \to \mathsf{List}\,A$, a proof that p satisfies the equality can always be constructed. For instance the proof for $p = \mathsf{swapHeads}$ would look similar to the proof swapHeadsNat-map explained in Section 1.1.1 (this is due to the fact that $\mathsf{swapHeads}\,\mathbb{N} = \mathsf{swapHeadsNat}$). In summary, as Wadler [Wad89] puts it: "From the type of a polymorphic function we can derive a theorem it satisfies. Every function of the same type satisfies the same theorem."

The above analysis about polymorphic programs was performed intuitively and informally. By contrast, Reynolds [Rey83] was the first to rigorously define the notion of uniformity/parametricity for polymorphic functions. We first give the idea behind his definition and then explain why it entails theorems for free.

1.2.3 Relational Parametricity

According to Reynolds, a polymorphic function $p : (A : \mathsf{Type}) \to T\,A$ is *parametric* if p maps related inputs to related outputs. A bit more precisely, p is parametric if it maps any relation[1] $R : A_0 \to A_1 \to \mathsf{Type}$ between input types to a proof that the outputs $p\,A_0 : T\,A_0$ and $p\,A_1 : T\,A_1$ are "related". The latter notion of relatedness between outputs cannot straightforwardly be expressed with R because the outputs of p are not of type A_0, A_1 but rather of type $T\,A_0, T\,A_1$. Hence what is meant by "related" outputs must take into account the structure of the dependent type T, and Reynolds's definition of parametricity does specify what it means for terms to be T-structurally related, for each kind of type T (the notion of T-structural relatedness is expressed with the notation $[T]_2$ below). Consequently a rigorous definition of Reynolds's parametricity is necessarily lengthy and we give one later. Note that Reynolds's definition is commonly referred to as relational parametricity, or simply as parametricity in the literature and was initially formulated for functions expressible in the System F language [Gir72; Gir86; Rey74]. The latter is a minimalist language featuring polymorphism.

As an example, the statement that swapHeads is relationally parametric is the $[\mathsf{swapHeads}]_2$ theorem below. Note that in Agda identifiers may contain symbols and $[\mathsf{swapHeads}]_2$ is a correct identifier. The bracket notation for parametricity statements is common in the literature, and the index 2 refers to the fact that *binary* relations are considered.

```
[List]₂ : {A0 A1 : Type} → (R : A0 → A1 → Type) → List A0 → List A1 → Type
[List]₂ R [] [] = Unit
[List]₂ R [] (a1 :: a1s) = ⊥
[List]₂ R (a0 :: a0s) [] = ⊥
```

[1] The type of R can be understood as the type of statements depending on inputs in A_0, A_1.

```
[List]₂ R (a0 :: a0s) (a1 :: a1s) = R a0 a1 × ([List]₂ R a0s a1s)

-- recall that swapHeads : (A : Type) → List A → List A
[swapHeads]₂ : ∀ A0 A1 → (R : A0 → A1 → Type) →
    (a0s : List A0) (a1s : List A1) → [List]₂ R a0s a1s →
    [List]₂ R (swapHeads A0 a0s) (swapHeads A1 a1s)
[swapHeads]₂ A0 A1 R [] [] hyp = tt
[swapHeads]₂ A0 A1 R (a0 :: []) (a1 :: []) hyp = hyp
[swapHeads]₂ A0 A1 R (a0 :: a0' :: a0s) (a1 :: a1' :: a1s) hyp =
    hyp .snd .fst , hyp .fst , hyp .snd .snd
```

To understand $[\mathsf{swapHeads}]_2$, we first review $[\mathsf{List}]_2$. The $[\mathsf{List}]_2$ type former defines a notion of structural relatedness for lists $a0s : \mathsf{List}\, A_0$ and $a1s : \mathsf{List}\, A_1$. Given a relation $R : A_0 \to A_1 \to \mathsf{Type}$, it asserts that the lists are related $[\mathsf{List}]_2\, R\, a0s\, a1s$ if and only if they have the same length and each of their corresponding entries are related by R. Technically the definition of $[\mathsf{List}]_2$ mentions the singleton type Unit (i.e. the true statement), the empty type $\bot$ (i.e. the false statement), and a product type $\times$ (logical conjunction). Furthermore it is recursively defined, as $[\mathsf{List}]_2$ appears in its own definition, and total or terminating, as are all Agda programs. From $[\mathsf{List}]_2$, the relational parametricity statement $[\mathsf{swapHeads}]_2$ is defined. It asserts that given a relation R and two input lists $a0s$ and $a1s$ that are structurally related over R, the lists outputted by swapHeads are structurally related over R as well. In short $[\mathsf{swapHeads}]_2$ expresses that swapHeads *preserves structural relatedness*. The proof of $[\mathsf{swapHeads}]_2$ is performed by an induction mimicking the definition of swapHeads. This is not coincidental and is discussed later. Regarding this proof, note that clauses containing patterns from which Agda can infer a proof abs : $\bot$ can be omitted. The reason is that proving any statement in a context containing an absurd hypothesis is anyway trivial.

Relational parametricity is an appropriate mathematical definition of what parametric polymorphism is. Indeed, if p is relationally parametric, then one can derive free theorems for p in the aforementioned sense. For instance, from relational parametricity it is possible to prove the equality $\mathsf{map}\, f\, (\mathsf{swapHeads}\, A\, as) \equiv \mathsf{swapHeads}\, B\, (\mathsf{map}\, f\, as)$. The proof invokes $[\mathsf{swapHeads}]_2$ for an input relation $R : A \to B \to \mathsf{Type}$ set to be the graph of the function f.

In summary, the relational parametricity of a polymorphic function p is a proof $[p]_2$ that p preserves structural relations, and such a preservation property may be seen as a rigorous definition of what it means to behave uniformly. Moreover free theorems may be deduced from this property.

Historically and naturally, the parametricity mapping $[-]_2$ has been defined as an operation consuming expressions of a given object language as inputs, e.g. expressions of System F for Reynolds. Such parametricity mappings are called *external* in the literature since they are defined outside of the object language, e.g. not as a System F function. These mappings are discussed in the next section. By contrast, there exists

a more recent set of works that aim to extend the object language itself with, roughly, a primitive function $[-]_2$. These works are thus known as *internal* parametricity techniques and are discussed later in Section 1.4 (this thesis makes contributions to internal parametricity).

As a side note, external and internal parametricity techniques are here reviewed as additions to *total* languages, i.e. languages in which every program terminates. By contrast, there exists a number of works [MP88; AP90; Pit00] attempting to extend parametricity principles to languages with general recursion, with the aim of recovering free theorems and representation independence results in the presence of non-termination.

1.3 External parametricity

A principled and modern way to understand and write Reynolds's definition is to phrase it in terms of reflexive graphs (an idea that remains valid in internal approaches to parametricity). A reflexive graph is simply a directed graph where every vertex has an edge to itself, called the reflexivity edge. More precisely, it is a type A equipped with a type former of edges written $[A]_2 : A \to A \to \mathsf{Type}$ such that any term $a : A$ has a reflexivity edge $[a]_2 : [A]_2\, a\, a$.

Then, parametricity can in fact be phrased as the slogan: "types are reflexive graphs". More precisely parametricity can be thought of as a mapping, or "translation", equipping each type A with a certain reflexive graph structure over A. That is, the parametricity translation of a type A outputs a certain relation $[A]_2 : A \to A \to \mathsf{Type}$ which satisfies $\forall a.\, [A]_2\, a\, a$. Importantly $[A]_2 : A \to A \to \mathsf{Type}$ is of course not just any reflexive relation. It is defined in such a way that it corresponds exactly to a notion of A-structural relatedness. In other words $[A]_2\, a_0\, a_1$ is a statement specifying what it means for a_0 and a_1 to be structurally related as terms of their type A (see $[\mathsf{List}]_2$ above).

We will define $[-]_2$ on the types of DTT and see how it entails relational parametricity as discussed in Section 1.2.3. But first we make a few historical remarks about external parametricity techniques.

1.3.1 Historical digression

The connection between parametricity and reflexive graphs was first uncovered in [RR94], where Robinson and Rosolini provide a denotational model based on reflexive graphs for the system F language evoked above. In other words and roughly speaking, each type of System F gets mapped to a reflexive graph, and terms are interpreted as morphisms of reflexive graphs. System F is a minimalist language that has no notion of relations between types, thus relational parametricity of any polymorphic function f cannot be expressed in the language. Yet the interpretation

Table 1.1: Inputs and outputs of the parametricity translation for different works on external parametricity.

Work	Input type	Output type
[Rey83]	A : SysF	isRGraph(A)
[RR94]	SysF	RGraph
[BJP12]	A : DTT	isRGraph(A)
[AGJ14]	DTT	RGraph

of f does satisfy relational parametricity because this interpretation is a (dependent) morphism of graphs which preserves edges.

The significance of such a model for System F is that the user of the language can soundly assume that polymorphic functions are parametric. This makes it possible, for instance, to faithfully encode data types as types of polymorphic functions: a function $f : \forall A.\, A \to A \to A$ can not inspect its type argument A and is thus either $\lambda A\, a_1\, a_2.a_1$ or $\lambda A\, a_1\, a_2.a_2$. Therefore $\forall A.\, A \to A \to A$ is morally isomorphic to the data type of booleans, a type which is not available as a primitive in System F. Note that such encodings of data types are known in the literature as Böhm-Berarducci [BB85] encodings, or a bit less precisely as Church encodings (Church worked in untyped lambda-calculus).

Later on, by contrast with the Reynolds translation discussed above and the Robinson and Rosolini model which both operate on System F, Takeuti [Tak01], Bernardy et. al [BJP12] and Atkey et. al [AGJ14] extended the parametricity mapping to the types and terms of dependent type theory (DTT). DTT is indeed a generalization of System F since DTT additionally features types that can have term variables. In other words[2] (1) System F can express this type: $A : \mathsf{Type} \vdash A \to A\ \mathsf{type}$, since it is a type that features a variable standing for a type but (2) System F can not express the type: $a_0 : A, a_1 : A \vdash a_0 \equiv a_1\ \mathsf{type}$, since a_0, a_1 are variables that stand for *terms* of A.

Table 1.1 informally summarizes the parametricity mappings provided by some of the works cited so far. Each mapping may or may not be a dependent function. Some mappings are dependent since they output a reflexive graph structure over A (see isRGraph predicate in the table). And some mappings are not dependent since they output a full-on reflexive graph (whose carrier is basically A).

1.3.2 Typing rules of DTT

In order to be able to define the parametricity translation on the types of DTT, we first need to review them.

[2] $\Gamma \vdash B\ \mathsf{type}$ is a judgment saying "B is a well-formed type that may mention variables from Γ".

$$\frac{}{\cdot\ \mathsf{ctx}}\ \textsc{emp} \qquad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A\ \mathsf{type}}{\Gamma,\ x : A\ \mathsf{ctx}}\ \textsc{cext}$$

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type}\ \mathsf{type}}\ \textsc{Typefm}$$

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma,\ x : A \vdash B\ \mathsf{type}}{\Gamma \vdash (x : A) \to B\ \mathsf{type}}\ \Pi\textsc{fm} \qquad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma,\ x : A \vdash B\ \mathsf{type}}{\Gamma \vdash (x : A) \times B\ \mathsf{type}}\ \Sigma\textsc{fm}$$

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash \mathsf{List}\ A\ \mathsf{type}}\ \textsc{Listfm} \qquad \frac{(g_1 : G_1, \ldots, g_m : G_m)\ \mathsf{ctx}}{g_1 : G_1, \ldots, g_m : G_m \vdash g_k : G_k}\ \textsc{var}$$

$$\frac{\Gamma,\ x : A \vdash b : B}{\Gamma \vdash \lambda\, x.\, b : (x : A) \to B}\ \Pi\textsc{i} \qquad \frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\left\{\frac{a}{x}\right\}}\ \Pi\textsc{elim}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B\left\{\frac{a}{x}\right\}}{\Gamma \vdash (a,\ b) : (x : A) \times B}\ \Sigma\textsc{i} \qquad \frac{\Gamma \vdash p : (x : A) \times B}{\Gamma \vdash \mathsf{fst}\, p : A}\ \Sigma\textsc{e}_1$$

$$\frac{\Gamma \vdash p : (x : A) \times B}{\Gamma \vdash \mathsf{snd}\, p : B\left\{\frac{\mathsf{fst}\, p}{x}\right\}}\ \Sigma\textsc{e}_2 \qquad \frac{\Gamma,\ x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda\, x.\, b)\, a = b\left\{\frac{a}{x}\right\} : B\left\{\frac{a}{x}\right\}}\ \Pi\beta$$

$$\frac{\Gamma, a : A \vdash f_0\, a = f_1\, a : B\left\{\frac{a}{x}\right\}}{\Gamma \vdash f_0 = f_1 : (x : A) \to B}\ \Pi\eta \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B\left\{\frac{a}{x}\right\}}{\Gamma \vdash \mathsf{fst}\,(a,\ b) = a : A}\ \Sigma\beta_1$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B\left\{\frac{a}{x}\right\}}{\Gamma \vdash \mathsf{snd}\,(a,\ b) = b : B\left\{\frac{a}{x}\right\}}\ \Sigma\beta_2$$

$$\frac{\Gamma \vdash \mathsf{fst}\, p_0 = \mathsf{fst}\, p_1 : A \qquad \Gamma \vdash \mathsf{snd}\, p_0 = \mathsf{snd}\, p_1 : B\left\{\frac{a}{x}\right\}}{\Gamma \vdash p_0 = p_1 : (x : A) \times B}\ \Sigma\eta$$

Figure 1.1: Some basic rules of DTT. Some premises are omitted. Substituting a variable x by a term a is expressed as $\left\{\frac{a}{x}\right\}$.

DTT is a language whose well-typed expressions (types, terms, contexts, substitutions) are defined by mutual recursion. Such expressions can be formed via inference rules. Some of these rules are listed in Fig. 1.1 and briefly explained below. It is best to think of these rules as a precise but somewhat informal specification of what it means to be well-formed and well-typed. The greater issue of formally defining the syntax of DTT [BHL20] and, e.g., specifying it to a computer [Nor07] is far from trivial.

The conclusion of each rule in Fig. 1.1 is a judgment asserting that a certain expression is well-formed and well-typed (well-typed for short). Four kinds of judgments appear in Fig. 1.1: judgments asserting that a context is well-typed $\Gamma\ \mathsf{ctx}$, that a type is well-typed $\Gamma \vdash A\ \mathsf{type}$, that a term is well-typed $\Gamma \vdash a : A$, that two terms are definitionally equal $\Gamma \vdash a_0 = a_1 : A$ (an example of a derivation using this equality appeared in Section 1.1.1). The judgment concluded by a rule may be

predicated by other judgments called premises appearing above the line. And premises may be judgments of a different kind than the conclusion, as in CEXT. In other words the different kinds of judgments are mutually recursively defined (in a well-founded way, it turns out).

We comment on some of the rules for contexts and types since $[-]_2$ is defined on these in what follows. The context extension rule CEXT conveys the idea that contexts are lists of well-formed types. An example of a context of size two is $(\cdot, A : \mathsf{Type}, f : (B : \mathsf{Type}) \to \mathsf{Type})$ (several rules are used to produce it, in particular CEXT is used twice). Of course $\cdot$ is usually omitted. It is quite important to note that here A and f *are not terms*, that is, are not expressions produced with the rules of the language. They are rather notational entities called free variables which can be referred to in terms and types to the right of $\vdash$. Such references are in fact constructed with the VAR rule. Moreover, a convention is that all the variables in a context Γ are distinct. By contrast the types appearing in a context *are* well-formed expressions according to the rules of DTT, and this is what CEXT expresses.

The ΠFM rule is the rule to form dependent function types, also known as Π-types. This rule asserts that this type former expects as inputs (1) a type $\Gamma \vdash A$ type that may mention free variables from Γ and (2) a type $\Gamma, x : A \vdash B$ type that may mention an extra free variable x, standing for something of type A. If two such types are provided then $(x : A) \to B$ is a notation for a new type which may mention free variables in Γ. Importantly, the references to x that may have appeared in the type $\Gamma, x : A \vdash B$ are now understood to point to the binding $(x : A) \to$. Note that function types $A \to B$ are obtained as particular cases of dependent function types: $A \to B := (x : A) \to B$.

The ΣFM rule defines a primitive type former of dependent pairs, also called a Σ-type. Dependent pairs are pairs (a, b) in which the type of b may depend on a. Note that product types $A \times B$ are obtained as particular cases of dependent pairs types: $A \times B = (x : A) \times B$.

Overall the well-typed expressions of DTT are obtained by combining together primitive typing rules such as the ones displayed in Fig. 1.1. So the proof that a certain composite expression is well-typed, e.g. $\Gamma \vdash a : A$, will be a derivation tree using the rules for the primitives in a.

1.3.3 Parametricity translation

We provide a partial definition of the parametricity translation $[-]_2$ on DTT following [BJP12]. The partial definition explains what $[A]_2$ is for some type formers of DTT, and suffices for our purposes. Relational parametricity as discussed in Section 1.2.3 is recovered afterwards by computing $[A]_2$ for A a particular type of polymorphic functions.

Providing a complete definition is possible but cumbersome for a number of reasons, which we invite the reader to reflect on. First $[-]_2$ is a dependent mapping:

from a type/term of DTT, $[-]_2$ outputs a relation/proof whose type depends on the input type/term. Second, $[-]_2$ is defined by induction, i.e. by performing a case analysis on its input type/term. So $[-]_2$ must be defined differently on each type former of DTT. What's more, types of DTT may depend on terms so strictly speaking one can not define $[-]_2$ for types in isolation, as there are certain types A for which $[A]_2$ must know how to compute on term formers. Third, $[-]_2$ operates on dependent type theory and the types of DTT are of the form $\Gamma \vdash A$ type and can be open, i.e. mention free variables from their surrounding context Γ (this is also true for System F). Hence, fourth, the translation must be defined on contexts as well. Overall $[-]_2$ is a mapping defined by induction on DTT, and this gets involved because DTT is a mutually recursively defined language specifying valid types, terms, contexts.[3] For the expert reader, we note that a principled way to provide a complete definition for $[-]_2$ is to turn the mapping $A \mapsto \mathsf{isRGraph}(A)$ into what is known as a displayed model [Kap25; BKS23] over DTT (e.g. a displayed category with families). Then $[-]_2$ emerges as the unique section of that model.

A definition of the binary parametricity translation appears in Fig. 1.2 and we make some technical remarks about it. The intuition behind the definition is provided afterwards. Regarding contexts, as can be seen from the context translation, parametricity is a context-growing operation. A context $\Gamma = (g_1 : G_1, ...g_m : G_m)$ of size m gets translated to a context $[\Gamma]_2$ of size $3m$. Indeed each g_k is replaced with two variables $g_{k,0}$ and $g_{k,1}$ standing for endpoints, as well as a variable g_k^r whose type expresses that $g_{k,0}$ and $g_{k,1}$ are related. This is the content of the clause for $[\Gamma, a : A]_2$. Regarding types, the first line in the type translation expresses that the translation $[A]_2$ of a type A in context Γ, more precisely written $[\Gamma \vdash A]_2$, is a relation between terms of A. Since the context has grown from Γ to $[\Gamma]_2$, one must substitute ($\{\}$ notation) the free variables of $\Gamma \vdash A$ with variables available in $[\Gamma_2]$. To that end the leftmost occurrence of A uses the "left" variables $g_{k,0}, 1 \leq k \leq m$ and the rightmost occurrence of A uses the "right" variables $g_{k,1}, 1 \leq k \leq m$. Regarding terms, the translation of a term a, written $[a]_2$ or more precisely $[\Gamma \vdash a : A]_2$, asserts that it is related to itself. A similar substitution must be performed as for types for the same reason. Indeed a lives in context Γ and the context has changed to $[\Gamma]_2$. A last minor technicality is that for explanatory purposes we wrote the translation in such a way that it converts judgments ($\Gamma \vdash A$ type) to judgments ($[\Gamma]_2 \vdash [A]_2 : A\{\ldots\} \to A\{\ldots\} \to \mathsf{Type}$), i.e. it maps well-formed types to (basically) *terms* of type Type. A proper definition should avoid such a mismatch.

The intuition behind the technical definition is that the translation on types defines a notion of structural relatedness, as summarized here ($[\mathsf{List}]_2$ was defined in Agda in

[3] And technically substitutions between contexts.

Context translation, where

$$\Gamma = (g_1 : G_1, \ldots, g_m : G_m)$$

$$[\cdot]_2 = \cdot$$

$$[\Gamma,\, a : A]_2 = [\Gamma]_2,\ \left(a_0 : A\left\{\tfrac{g_{1,0},\ldots,g_{m,0}}{g_1,\ldots,g_m}\right\}\right),\ \left(a_1 : A\left\{\tfrac{g_{1,1},\ldots,g_{m,1}}{g_1,\ldots,g_m}\right\}\right),\ \left(a^r : [\Gamma \vdash A]_2\, a_0\, a_1\right)$$

Type translation, where

$$[\Gamma]_2 \vdash [\Gamma \vdash A]_2\ :\ A\left\{\tfrac{g_{1,0},\ldots,g_{m,0}}{g_1,\ldots,g_m}\right\}\ \to\ A\left\{\tfrac{g_{1,1},\ldots,g_{m,1}}{g_1,\ldots,g_m}\right\}\ \to\ \mathsf{Type}$$

$$\begin{aligned}[\Gamma \vdash (a : A) \to B]_2 = \lambda\, f_0\, f_1.\ &\left(a_0 : A\left\{\tfrac{g_{1,0},\ldots,g_{m,0}}{g_1,\ldots,g_m}\right\}\right) \to \left(a_1 : A\left\{\tfrac{g_{1,1},\ldots,g_{m,1}}{g_1,\ldots,g_m}\right\}\right)\\ &\to\ (a^r : [\Gamma \vdash A]_2\, a_0\, a_1)\ \to\ [\Gamma,\, a{:}A \vdash B]_2\ (f_0\, a_0)\ (f_1\, a_1)\end{aligned}$$

$$\begin{aligned}[\Gamma \vdash (a : A) \times B]_2 = \lambda\, p_0\, p_1.\ &\left(e : [\Gamma \vdash A]_2\ (\mathsf{fst}\ p_0)\ (\mathsf{fst}\ p_1)\right)\\ &\times\ [\Gamma,\, a{:}A \vdash B]_2\left\{\tfrac{\mathsf{fst}\ p_0,\, \mathsf{fst}\ p_1,\, e}{a_0,\, a_1,\, a^r}\right\}\ (\mathsf{snd}\ p_0)\ (\mathsf{snd}\ p_1)\end{aligned}$$

$$[\Gamma \vdash \mathsf{Type}]_2 = \lambda\, A_0\, A_1.\ A_0 \to A_1 \to \mathsf{Type}$$

$$[\Gamma \vdash \mathsf{List}\ A]_2 = \lambda\, l_0\, l_1.\ [\mathsf{List}]_2\ [\Gamma \vdash A]_2\ l_0\ l_1$$

Term translation (only for terms that are free variables), where

$$[\Gamma]_2 \vdash [\Gamma \vdash a : A]_2 : [\Gamma \vdash A]_2\ a\left\{\tfrac{g_{1,0},\ldots,g_{m,0}}{g_1,\ldots,g_m}\right\}\ a\left\{\tfrac{g_{1,1},\ldots,g_{m,1}}{g_1,\ldots,g_m}\right\}$$

$$[\Gamma \vdash g_k : G_k]_2 = g_k^r$$

Figure 1.2: Binary parametricity translation of DTT.

Section 1.2.3):

$$\begin{aligned}
[(a : A) \to B]_2\, f_0\, f_1 &= f_0 \text{ and } f_1 \text{ map related inputs to related outputs} \\
[(a : A) \times B]_2\, p_0\, p_1 &= p_0 \text{ and } p_1 \text{ have both components related} \\
[\mathsf{Type}]_2\, A_0\, A_1 &= \text{there exists a relation } A_0 \to A_1 \to \mathsf{Type} \\
[\mathsf{List}A]_2\, l_0\, l_1 &= l_0, l_1 \text{ have the same length and related entries}
\end{aligned}$$

Furthermore we can recover the relational parametricity of the swapHeads function from Section 1.2.3. Indeed the translation tells us

$$\vdash [\ \mathsf{swapHeads}\ ]_2 : [\ \forall A.\, \mathsf{ListA} \to \mathsf{ListA}\ ]_2\ \mathsf{swapHeads}\ \mathsf{swapHeads}$$

and $[\,\forall A.\, \mathsf{ListA} \to \mathsf{ListA}\,]_2$ computes to the type given for the Agda program named $[\mathsf{swapHeads}]_2$ provided in Section 1.2.3 ($[\mathsf{swapHeads}]_2$ is an Agda identifier). The latter type expresses that swapHeads maps related lists to related lists. Note that in the typing judgment above there is no need to substitute any free variables since swapHeads is a closed term, i.e. has no free variables. Lastly, though not defined above, the term translation would compute $[\ \mathsf{swapHeads}\ ]_2 = [\mathsf{swapHeads}]_2$ and indeed the proof $[\mathsf{swapHeads}]_2$ mimics the way swapHeads is defined.

To summarize:

- Binary external parametricity has been defined for types of DTT.
- From a type A parametricity computes a reflexive graph structure over A.
- The computed relation $[A]_2$ defines a notion of structural relatedness for A.
- Useful information can be extracted from $[A]_2\, a\, a$ (theorems for free).

1.3.4 n-ary Parametricity

So far we discussed binary parametricity, the fact that types are *binary* reflexive graphs, i.e. a type A can be endowed with a binary relation $[A]_2 : A \to A \to \mathsf{Type}$. The notion can be straightforwardly generalized to arity n and n-ary parametricity expresses that types are n-ary reflexive graphs. A translation like Fig. 1.2 can be performed similarly. As discussed later, this thesis makes contributions to binary $n = 2$ and nullary $n = 0$ internal parametricity. We say a word about the arities 0 and 1.

Nullary parametricity Without any addition to DTT, nullary parametricity is uninteresting. Indeed a nullary reflexive graph is (1) a type A with (2) a type of "nullary edges" $[A]_0$ such that (3) every $a : A$ has an edge, i.e. there is a function $[-]_0 : A \to [A]_0$. And given any type A, the nullary translation defined as in Fig. 1.2 computes the following $[A]_0 = A$, and $[a]_0 = a$.

By contrast, in Chapter 3 we will (essentially) postulate an additional type in DTT called Nm which satisfies $[\mathsf{Nm}]_0 = 1 + \mathsf{Nm}$, i.e. its translation is a sum type. This leads to a type theory well-suited to formalize languages with binders.

Unary parametricity To illustrate the case of $n = 1$, consider the following example (where suc : $\mathbb{N} \to \mathbb{N}$ is the successor function).

```
recℕ : (A : Type) (z : A) (s : A → A) → ℕ → A
recℕ A z s 0 = z
recℕ A z s (suc m) = s (recℕ A z s m)
```

This program is called the recursion principle of natural numbers, as it explains how to define a function $\mathbb{N} \to A$ by recursion. In order to do so it is sufficient to provide a starting point z and a step function $s : A \to A$. Then $\mathsf{rec}\mathbb{N}\, A\, z\, s\, m$ applies the s function m times to z. Moreover any recursive function on natural numbers is ultimately defined in this way.

We can compute by hand the unary translation $[-]_1$ of the closed (no free variables) Agda program $\vdash \mathsf{rec}\mathbb{N}$, leading to the following type and program.

```
-- type translation
REC = (A : Type) (z : A) (s : A → A) → ℕ → A
[REC]₁ : REC → Type₁
[REC]₁ r = ∀ (A : Type) (P : A → Type)
  (z : A) (zdot : P z)
  (s : A → A) (sdot : ∀ a → P a → P (s a)) →
  ∀ n → P(r A z s n)

-- term translation
[recℕ]₁ : [REC]₁ recℕ
[recℕ]₁ A P z zdot s sdot zero = zdot
[recℕ]₁ A P z zdot s sdot (suc n) = sdot (recℕ A z s n) ([recℕ]₁ A P z zdot s sdot n)
```

The type $[\mathsf{REC}]_1\ \mathsf{rec}\mathbb{N}$ looks quite similar to the *induction* principle of natural numbers.[4] As a reminder, proofs by induction on $\mathbb{N}$ reduce the task of proving $\forall n.\, P\, n$ to the task of proving a base case (corresponding to zdot above) and an induction step (corresponding to sdot).

1.3.5 The Issue with External Parametricity

There are statements about polymorphic programs that seem true but which DTT alone cannot possibly prove, nor refute. As an example, consider the following statement, which is a simple generalization of the free theorem $\mathsf{map}\ f\ (\mathsf{swapHeads}\ A\ as) \equiv \mathsf{swapHeads}\ B\ (\mathsf{map}\ f\ as)$, which we proved earlier.

```
map-commute : (p : ∀ X → List X → List X) →
  (A B : Type) (f : A → B) (as : List A) →
  map f (p A as) ≡ p B (map f as)
map-commute p A B f as = ?
```

[4] Recovering the principle from $[\mathsf{REC}]_1\ \mathsf{rec}\mathbb{N}$ is not entirely direct.

This statement asserts that the commutation property holds not just for swapHeads but for any program p with the same type. An informal justification of map-commute was provided in Section 1.2.2: basically p can not inspect its type argument. Crucially the quantification on p is here performed *internal* to DTT. That is to say, we were able as a user of DTT/Agda to express this quantified statement *within* DTT, using a Π-type. Such a statement is referred to as a *global* free theorem in Chapter 2.

In fact the map-commute statement is logically independent from DTT: there exist no proofs of it and no proofs of its negation in DTT (the negation of a type T is defined to be the type $\neg T := T \to \bot$, where $\bot$ is the empty type). Briefly, we know this because: (1) a statement that is proved in DTT must hold in all models, by definition of what a denotational model is (briefly discussed in Section 1.3.1) (2) there are models in which the statement is true, like the reflexive graph model of [AGJ14] (3) there are models in which it is false, e.g. DTT + the law of excluded middle $\forall A.\ A + \neg A$ (see [Boo+16]) (4) therefore the statement admits no proof and no refutation. More concretely, one cannot provide a proof of $?$ above because p is merely a variable and we do not know how $p\,A\,as$ computes on the list as. In the case of swapHeads we knew this because of how this program was defined.

In order to be able to prove global free theorems like map-commute within the language, DTT has to be extended with new type-theoretical primitives.

1.4 Internal Parametricity

Internally parametric type theories [BM12; BM13; BCM15; CH21; Alt+24] (parametric type theories, PTTs for short) are languages extending DTT with primitives that enable parametricity. More precisely, these primitives are additional type formers and term formers which make it possible to express proofs for parametricity statements, even global free theorems like map-commute. We outline how this works in Cavallo and Harper's theory [CH21] – the CH theory for short, since their system is central to this thesis.

The CH theory (based on [BCM15]) postulates a type former called $\mathsf{Bridge} : (A : \mathsf{Type}) \to A \to A \to \mathsf{Type}$ whose intent is to represent the parametricity translation $[-]_2$ inside the theory.[5] This aim is fulfilled in the sense that the following bijections (technically HoTT equivalences) can be proved within the CH theory, among others. This is similar to how $[\mathsf{Type}]_2$ and $[A \to B]_2$ are defined in Fig. 1.2.

$$\mathsf{Bridge}\,\mathsf{Type}\,A_0\,A_1 \quad \simeq \quad (A_0 \to A_1 \to \mathsf{Type})$$
$$\mathsf{Bridge}\,(A \to B)\,f_0\,f_1 \quad \simeq \quad (\forall a_0 a_1.\,\mathsf{Bridge}\,A\,a_0\,a_1 \to \mathsf{Bridge}\,B\,(f_0\,a_0)(f_1\,a_1))$$

The first bijection relies on a primitive called Gel converting a relation (right-hand side) to a bridge (left-hand side). Similarly the second bijection relies on a primitive

[5] Technical detail: for simplicity Bridge is here presented as a well-typed term of (basically) the type Type instead of a well-formed type.

called extent converting a proof of the right-hand side into a proof of the left-hand side.

It is important to note that these bijections are not definitional equalities: one could imagine a rule in the type theory asserting that $(\mathsf{Bridge}\ \mathsf{Type}\ A_0\ A_1) = (A_0 \to A_1 \to \mathsf{Type})$ but this is not the case (other systems attempt to feature similar equations and it leads to other complications). In fact the Bridge type is axiomatized in a completely different fashion, inspired from another class of DTTs called cubical type theories [Coh+17].

Cubical type theory Cubical type theory (cubical TT) axiomatizes the *equality* type $\_ \equiv \_$ in a special way. The theory postulates a certain type I called the path interval, and two endpoints in it $i_0, i_1 : \mathsf{I}$. Then a proof of equality $p : a_0 \equiv a_1$ is basically a function $p : \mathsf{I} \to A$ that satisfies $p\,\mathsf{i}_0 = a_0$ and $p\,\mathsf{i}_1 = a_1$. So pictorially p is a segment, or *path* between a_0 and a_1 in A. For that reason equality types are alternatively called path types in cubical TT. Cubical TT features other primitives which make it possible to prove univalence $(A_0 \equiv A_1) \simeq (A_0 \simeq A_1)$. Univalence asserts that two types are provably equal $A_0 \equiv A_1$ if and only if they are isomorphic $A_0 \simeq A_1$. Univalence allows a form of representation independence to take place when reasoning about software or mathematics in type theory: two isomorphic implementations of an interface are provably indistinguishable.

Bridges The CH theory is in fact a cubical TT [AFH18] so features a certain interval type I used to construct equality proofs. More importantly for us, the CH theory features another interval BI used to construct *bridges*, and for this reason it is deemed interval-based as a parametric type theory. In more details, the CH theory postulates a bridge interval type BI and two endpoints in it $\mathsf{bi}_0, \mathsf{bi}_1 : \mathsf{BI}$. Then a bridge $q : \mathsf{Bridge}\,A\,a_0\,a_1$ is basically a function $q : \mathsf{BI} \to A$ that satisfies $q\,\mathsf{bi}_0 = a_0$ and $q\,\mathsf{bi}_1 = a_1$. So pictorially q is a segment between a_0 and a_1 in A.

Now, a form of internal parametricity can be formulated and proved as follows within the CH type theory (to be compared with the fact that $\forall a.\,[a]_2 : [A]_2\,a\,a$):

$$\vdash \mathsf{rfl} : (A : \mathsf{Type})(a : A) \to \mathsf{Bridge}\,A\,a\,a$$

The proof is the reflexivity bridge $\mathsf{rfl} = \lambda A\,a.\lambda(x : \mathsf{BI}).\,a$, which ignores its bridge variable x and outputs a constantly. The proof of this fact is so trivial that the statement may seem insignificant. However this result does lead to free theorems such as map-commute from Section 1.3.5.

1.4.1 An issue of interval-based PTTs

The reason why $\mathsf{rfl}\,A\,a : \mathsf{Bridge}\,A\,a\,a$ above leads to parametricity results is that for every[6] type former X of the CH theory, the primitives make it possible to show

[6]This quantification is external, i.e. not expressed in the CH theory.

a bijection $\mathsf{Bridge}\, X\, x_0\, x_1 \simeq \ldots$, like the ones given above, expressing that the Bridge type former and the X type former commute (for instance one of the above bijections says that the Bridge type of a function type is equivalent to a function type of Bridge types). Hence, the Bridge type former in $\mathsf{Bridge}\, A\, a\, a$ can be commuted with every primitive type former in A, ultimately resulting in a type which is actually $[A]_2\, a\, a$ (note: $[-]_2$ is not a type former, but rather a metatheoretical operation applied to the A expression). Finally, through this bijection $\mathsf{Bridge}\, A\, a\, a \simeq [A]_2\, a\, a$, the reflexivity bridge in $\mathsf{Bridge}\, A\, a\, a$ may be mapped to a formal proof of the more useful parametricity statement $[A]_2\, a\, a$.

This methodology of recovering actionable parametricity statements is impractical because it requires rote work of the user. Indeed the complexity of proving $\mathsf{Bridge}\, A\, a\, a \simeq [A]_2\, a\, a$ is at least the complexity of computing $[A]_2\, a\, a$, and this must be performed step-by-step by the user by chaining bijections together in the CH theory. Furthermore A may also contain term formers which further complicate the issue.

In summary, parametric type theories like the CH theory offer parametricity internally and validate global free theorems, contrary to the external approaches to parametricity of Section 1.3. But they are impractical for performing parametricity proofs because, basically, the Bridge type does not compute on A like $[-]_2$ would.

As an additional side note, internally parametric type theories provide what is called iterated parametricity, contrary to external approaches. The idea is that the internal parametricity mapping rfl is a term of the CH theory. As such, it is provably parametric. Indeed rfl proves that all terms are parametric (up to the impractical rote work described above). So the following holds:[7]

$$\begin{aligned} &\vdash \mathsf{rfl}\ ((A : \mathsf{Type})(a : A) \to \mathsf{Bridge}\, A\, a\, a)\ \mathsf{rfl} \\ &: \mathsf{Bridge}\ ((A : \mathsf{Type})(a : A) \to \mathsf{Bridge}\, A\, a\, a)\ \mathsf{rfl}\ \mathsf{rfl} \end{aligned}$$

This is not possible using external parametricity approaches because $[-]_2$ consumes expressions of an object language from which it is absent.

1.5 Nominal syntax

We review the nominal representation of syntax since a contribution of this thesis will be to relate this notion to nullary internal parametricity.

When studying a certain object language on paper it is a simplifying practice to think of variables as being literal strings and to think of expressions up to α-renaming. For instance $\lambda x.\, xx$ and $\lambda y.\, yy$ are two expressions of the untyped lambda-calculus (ULC) that are equal, following the latter convention. However, when working completely formally (e.g. when proving properties of ULC in a proof assistant) it

[7] Universe levels considerations are omitted.

is preferable to represent the syntax of the object language in such a way that α-equivalent terms are equal by construction, contrary to the variables-as-strings representation. For example representing variables with De Bruijn indices validates this property.

Nominal syntax, a.k.a nominal data types, is a formal representation combining the above advantages (and more), that is, (1) a human-readable representation where one can use something close to strings for variables ("names") and (2) satisfying the property that α-equivalent terms are equal by construction.

Nominal data types feature a special type former, not definable in plain type theory or Agda, called the name abstraction type former, as well as a type of names. This name abstraction type former is used to give a type to the binders of the object language. For example the syntax of ULC can be represented as the following nominal data type, where the name abstraction type former is written $@N \multimap -$, and the type of names is written Nm.

```
data Ltm : Type where
  var : Nm → Ltm
  app : Ltm → Ltm → Ltm
  lam : (@N ⊸ Ltm) → Ltm
```

The metalanguages that provide a name abstraction type former and where one can express nominal data types are called nominal frameworks [Pit03; SPG03; PMD14; SS04; Che12].

Some nominal frameworks [PMD14; Che12; SS04] axiomatize name abstraction in such a way that it behaves as a type of functions consuming fresh names. More precisely, $a' : @N \multimap A$ is thought of as a function expecting a "name" $x : @N$ and outputting some $a : A$, with the extra restriction that $x : @N$ should not already appear in a' freely. For instance the following term does not typecheck since the freshness side condition does not hold.

```
diag : (a'' : @N ⊸ (@N ⊸ A)) → @N ⊸ A
diag a'' = λ (x : @N). (a'' x) x
```

The problem here is that $a'' \, x$ contains x as a free variable thus $a'' \, x$ is not allowed to consume the name x again, i.e. the example does not typecheck. Note that functions consuming fresh names are also called affine, or simply fresh functions.

Since $@N \multimap \mathsf{Ltm}$ is basically a function type, the ULC term $\lambda x.\, x \, x$ can be written $\mathsf{lam}\,(\lambda(x : @N).\, \mathsf{app}\,(\mathsf{var}\,x)\,(\mathsf{var}\,x))$.[8] It is the case that the latter is equal to $\mathsf{lam}\,(\lambda(y : @N).\, \mathsf{app}\,(\mathsf{var}\,y)\,(\mathsf{var}\,y))$, by the typing rules of $@N \multimap -$. This shows that nominal data types constitute a human-readable representation of syntax where α-equivalent terms are equal by construction.

[8] Ignoring the mismatch between $@N$ and Nm.

1.5.1 An issue with nominal frameworks

This representation is natural, yet existing nominal frameworks with function-like name abstraction $@N \multimap -$ fall critically short in expressivity (other nominal frameworks axiomatizing $@N \multimap -$ in a different way exist but have other issues, as discussed in Chapter 3). Indeed, such frameworks fail to express even routine operations on nominal data types, such as Ltm.

The basic reason is that $@N \multimap -$ is a type former of functions and as such does not provide a way to pattern match. As an example, the definition of this operation computing free names of an object term can not be completed.

```
fns : Ltm → List Nm
fns (var n) = n :: []   -- singleton list
fns (app a b) = fns a ++ fns b   -- concatenation
fns (lam t') = ?   -- t' : @N ⊸ Ltm
```

1.6 Contributions

This thesis makes two classes of contributions to internal parametricity. The first class is concerned with *binary* internal parametricity in Chapters 2 and 4. The second class is concerned with *nullary* internal parametricity and its relation to the study of syntax with binders in Chapter 3. An overview of the contributions is given here. More detailed lists of contributions appear in each chapter.

1.6.1 Binary Internal Parametricity

In order to be able to explore the consequences of parametricity and delegate its heavy bookkeeping aspects to the computer, it is desirable to have a typechecker, or even better, a full-fledged proof assistant implementation of parametric type theory. When this thesis began there only existed experimental or partial such implementations [Cav20; NVD17].[9]

Agda --bridges: a parametric proof assistant We contribute Agda --bridges [VND23; VND24], an extension of the Agda proof assistant that implements the interval-based parametric type theory of Cavallo and Harper [CH21]. The CH theory is in particular a cubical type theory [AFH18], meaning that proofs of equality are treated as paths from an interval I. Likewise Agda --bridges extends Agda --cubical [VMA21], a proof assistant implementing cubical TT. Hence useful theorems provided by Agda --cubical like univalence are available in Agda --bridges. Additionally Agda --bridges typechecks the standard library of Agda --cubical, as well as important theorems for

[9] There now also exists Narya [Nar25a], a WIP implementation of observational parametric type theory [Alt+24].

internal parametricity such as the equivalence $\mathsf{Bridge}\ \mathsf{Type}\ A_0\ A_1 \simeq (A_0 \to A_1 \to \mathsf{Type})$, called relativity.

ROTT: an Agda --bridges library suppressing the rote work An issue discussed in Section 1.4.1 is that recovering actionable parametricity statements and free theorems requires rote work, since there is a mismatch between bridge types and the parametricity translation. That is, this correspondence $\mathsf{Bridge}\ A\ a_0\ a_1 \simeq [A]_2\ a_0\ a_1$ is only a bijection that must be established step-by-step by the user of the theory. Our solution to this problem is to turn this mismatch into a definition. More precisely, we define a "relativistic reflexive graph" to be a type equipped with such an equivalence (the name comes from the fact that relativity is the relational analogue of univalence in [CH21]). Then the user can (1) express the desired type A as a relativistic reflexive graph using our domain-specific language/library called ROTT (2) extract a proof of an actionable (by contrast with rfl in Section 1.4.1) parametricity statement by invoking the param function from our ROTT library. Such invocations are usually one-liners. This method to obtain free theorems within Agda --bridges is compositional, user-friendly, and concise.

Theorems for free, global, typechecked The following theorems are formalized in Agda --bridges, following the above method.[10] The (global) map-commute theorem. A family of Church encodings, For instance $\mathsf{Bool} \simeq (\forall A.\ A \to A \to A)$; there is also a Church encoding for untyped lambda terms (using De Bruijn variables) as well as a Church encoding of the Vec indexed type of lists of fixed length. A parametric model of System F (on types): each type of System F is mapped to a relativistic reflexive graph; this establishes a connection between Reynolds's original definition and internal parametricity. Robin Schepens [Sch+24] was able to formalize results about effect parametricity in Agda --bridges: free theorems about polymorphic functions quantifying on monads [Voi09].

Adequacy of (P)HOAS for free In addition to the previous free theorems, in Chapter 4 we transcribe in Agda --bridges a proof that was communicated to us by Andrea Vezzosi, and expand on it. This results in a proof, in particular, that the following encodings are both isomorphic to the type of closed untyped lambda terms (defined with De Bruijn indices).

$$\mathsf{PhoasEnc} = \forall V\ A.\ (V \to A) \to (A \to A \to A) \to ((A \to A) \to A) \to A$$
$$\mathsf{HoasEnc} = \forall A.\ (A \to A \to A) \to ((A \to A) \to A) \to A$$

Previous known proofs [Atk09] used Kripke binary parametricity, a notion strictly stronger than binary parametricity.

[10] See https://github.com/antoinevanmuylder/bridgy-lib/tree/main/Bridgy/Examples

1.6.2 Nullary Internal Parametricity and Nominal frameworks

We saw in Section 1.5.1 that, in existing nominal frameworks with function-like name abstractions $@N \multimap -$, even routine operations can not be defined on nominal data types.

A nominal framework rooted in parametricity We contribute in Chapter 3 an improved nominal framework called Parametric Nominal Type Theory (PNTT) which fixes this issue, based on the internally parametric type theory of CH [CH21] with the arity set to 0. Concisely put: PNTT = nullary CH theory + a type Nm + a novel *name induction* principle for Nm + rules for nominal data types. PNTT is a good candidate for a principled implementation of a nominal framework, in fact this thesis implemented the related binary CH theory (Section 1.6.1). More details follow.

Nullary CH, read nominally We recognize that the nullary CH theory on its own can be seen as a nominal framework with several important features.

- The bridge type former is name abstraction $@N \multimap -$ and the nullary interval is $@N$.
- The nullary Gel primitive (the binary version was mentioned in Section 1.4) of nullary CH is read as a type former expressing freshness. Specifically, $\mathsf{Gel} : \mathsf{Type} \to (@N \multimap \mathsf{Type})$ and for a type A and bridge variable x, the type $\mathsf{Gel}\, A\, x$ can be read as "the a's in A in which x is fresh".
- Other primitives of nullary CH allow to implement primitives of various nominal frameworks.

Key result The reason why PNTT is able to recover from the issue discussed in Section 1.5.1 is ultimately that $@N \multimap -$ commutes with any type former in a specific way (similar to Section 1.4.1 for arity 2), *including* Nm. For the latter type we prove $(@N \multimap \mathsf{Nm}) \simeq 1 + \mathsf{Nm}$, using our name induction principle. To see why this fixes the issue in Section 1.5.1, observe that the recursion principle of Ltm entails that providing a term $?'$ as in the diagram below is enough to obtain a the missing clause $?$ in Section 1.5.1 (here $@N \multimap \mathsf{fns} := \lambda t'.\, \lambda(x : @N).\, \mathsf{fns}\,(t'\, x)$).

$$\begin{array}{ccc} @N \multimap \mathsf{Ltm} & \xrightarrow{@N \multimap \mathsf{fns}} & @N \multimap \mathsf{List}\,\mathsf{Nm} \\ {\scriptstyle \mathsf{lam}}\downarrow & & \downarrow{\scriptstyle ?'} \\ \mathsf{Ltm} & \xrightarrow[\mathsf{fns}]{} & \mathsf{List}\,\mathsf{Nm} \end{array}$$

Providing $?'$ *can* be done because the domain of this function can be characterized as follows $(@N \multimap \mathsf{List}\,\mathsf{Nm}) \simeq (\mathsf{List}\,(@N \multimap \mathsf{Nm})) \simeq (\mathsf{List}(1 + \mathsf{Nm}))$. Hence providing $?'$ is equivalent to providing $?'' : \mathsf{List}(1 + \mathsf{Nm}) \to \mathsf{List}(\mathsf{Nm})$ and we can define $?''$ to be the map forgetting entries that are in the left summand. Overall the

map ? obtained using this methodology is in fact correct, i.e. computes free names as expected. Interestingly, proving this fact requires characterizing $@N \multimap \mathsf{fns}$ as the parametricity translation $[\mathsf{fns}]_0$.

1.7 Outline

Regarding binary parametricity:

- The exact type system that underlies Agda --bridges is described in Section 2.2.
- The ROTT library and the issues it solves are described in Section 2.3.
- Section 2.4 describes how free theorems can be proved in Agda --bridges using ROTT.
- Section 2.5 explains how Agda --bridges integrates with Agda --cubical.
- Section 2.6 gives a comprehensive account of related work in external and internal parametricity.
- Chapter 4 uses Agda --bridges to formalize a proof that the PHOAS and HOAS representations are adequate.

Regarding nullary parametricity and nominal frameworks:

- In Section 3.2 the rules of Parametric Nominal Type Theory are presented and core theorems are proved.
- In Section 3.3 primitives from other nominal frameworks are reimplemented with primitives from Parametric Nominal Type Theory.
- In Section 3.4 we explain how Parametric Nominal Type Theory recovers nominal pattern matching by analysing two distinct object languages (see Section 3.4.1 and Section 3.4.2).

We conclude in Chapter 5.

Chapter 2

Internal and Observational Parametricity for Cubical Agda

This chapter is an exact reproduction of the following published paper, up to minor cosmetic changes.

> A. Van Muylder, A. Nuyts, and D. Devriese. "Internal and Observational Parametricity for Cubical Agda". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 209–240. DOI: 10.1145/3632850. URL: https://doi.org/10.1145/3632850

The research presented in this chapter was carried out by me, with supervision and contributions from Dominique Devriese and Andreas Nuyts. They helped shape the ideas presented in this chapter and contributed to the writing of the text. For the implementation of Agda --bridges, I also received early guidance from Andrea Vezzosi, a developer of Agda --cubical [VMA21].

2.1 Introduction

Theorems for free [Wad89] 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).

$$p : \forall\{X : \mathsf{Type}\} \to \mathsf{List}\, X \to \mathsf{List}\, X \to \mathsf{List}\, X \tag{2.1}$$

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:

$$\begin{gathered}\forall(A_0\, A_1 : \mathsf{Type})(xs\, ys : \mathsf{List}\, A_0)(f : A_0 \to A_1) \to \\ \mathsf{map}\, f\, (p\, xs\, ys) \equiv p\, (\mathsf{map}\, f\, xs)\, (\mathsf{map}\, f\, ys)\end{gathered} \tag{2.2}$$

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 [Rey83] for System F, a.k.a. the second-order polymorphic lambda calculus [Gir72; Gir86; Rey74]. 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:

$$\begin{gathered}\forall A_0\, A_1\, (R : \mathsf{Rel}(A_0, A_1)) \to \forall\, xs_0\, xs_1 \text{ such that } \mathsf{ListRel}_R\, xs_0\, xs_1 \to \\ \forall\, ys_0\, ys_1 \text{ such that } \mathsf{ListRel}_R\, ys_0\, ys_1 \to \mathsf{ListRel}_R\, (p\, xs_0\, ys_0)\, (p\, xs_1\, ys_1)\end{gathered} \tag{2.3}$$

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 (2.2) by setting $R(x, y)$ iff $f\, x \equiv y$ in property (2.3).

Reynolds's abstraction theorem is a metatheoretical statement about programs of System F. When verifying polymorphic programs in dependently typed proof assistants like Coq [Coq22] or Agda [Agd23], such a metatheoretical statement is

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

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 [BJP12; KL12; AM17] 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 (2.3) 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 (2.3).

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

$$\begin{aligned}\mathsf{glob\text{-}fthm} : \forall(p : \{X : \mathsf{Type}\} \to \mathsf{List}\, X \to \mathsf{List}\, X \to \mathsf{List}\, X)\\ \forall(A_0\, A_1 : \mathsf{Type})(xs\, ys : \mathsf{List}\, A_0)(f : A_0 \to A_1) \to\\ \mathsf{map}\, f\, (p\, xs\, ys) \equiv p\, (\mathsf{map}\, f\, xs)\, (\mathsf{map}\, f\, ys)\end{aligned} \tag{2.4}$$

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 [BPT17; Boo+16]. 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* [NVD17; ND18; Mou16; CH21]. In such type theories, free theorems can be proven from first principles, even theorems like (2.4). Those type theories draw inspiration from parametric denotational models of DTT (see e.g. [AGJ14]) 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. [Cav20; NVD17]). More fundamentally, even proving simple free theorems in those systems can be cumbersome. For example, Cavallo and Harper [CH21] 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 [VMA21]. It implements the parametricity primitives of Cavallo and Harper [CH21] (CH), which we have adapted to the cubical type theory underlying Agda --cubical [Coh+17]. 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 [MM18] in Agda --guarded [VV20; VV23] 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. [RWL22] for some background and references). Agda --bridges successfully compiles the full Agda --cubical standard library [Agd], 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 [CH21], 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 (2.4) 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. [Ang+21b] and Section 2.6.3). 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 (2.1), 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 (2.3):

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

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 2.3.3.2), 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 1) and the useful fact that the proof-irrelevant fragment of a type can be ignored while proving the SIP for it (Theorem 2).

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) [AK15; AKS22; PT22; AMS07] 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 (2.4); 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 [NVD17]. 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 [CH21] (CH) to the type theory underlying Agda --cubical [Coh+17] (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 [MM18] in Agda --guarded [VV20; VV23]. 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 [Agd] 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 2.2 we present the implementation and typing rules of Agda --bridges as well as its core theorems. More precisely: Section 2.2.1 is a brief reminder about cubical type theory and Agda --cubical; Section 2.2.2 discusses what makes Agda --bridges and the CH type theory substructural type systems. Sections 2.2.3, 2.2.4, 2.2.5 introduce the BridgeP, extent and Gel parametricity primitives; Section 2.2.6 and Section 2.2.7 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 2.5 as they are not needed for the free theorems we present. In Section 2.3 we present a framework developed in Agda --bridges that provides abstractions to write user-friendly, modular and concise proofs of free theorems[2].

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

More precisely: In Section 2.3.1 we discuss the structure relatedness principle (SRP); In Section 2.3.2 we identify obstructions to the SRP; In Section 2.3.3 we present ROTT and its param rule. We use ROTT and apply param on various examples in Section 2.4. We conclude with related work in Section 2.6.

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 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 $\{0, 1\}$.

2.2 The Internal Parametricity of Agda Bridges

Agda --bridges is a proof assistant extending the Agda --cubical proof assistant [VMA21]. 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 [Coh+17]). A reminder about Agda --cubical appears in Section 2.2.1.

The type theory of Agda --bridges is an adaptation of the internally parametric DTT of Cavallo and Harper [CH21] (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 2.2.2. 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 2.2.3. 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 2.5.

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 [HRR13], 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 2.2.3, 2.2.4, 2.2.5. Occurrences of these primitives in a program or type may generate freshness constraints at typechecking time.

In Section 2.2.6 and Section 2.2.7 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.

2.2.1 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 2.2.1.1). Second, it features language primitives that let it realize *univalence* as a theorem (see Section 2.2.1.3). The latter property characterizes type equality as type equivalence (i.e. having a "bijection") and constitutes a defining aspect of *homotopy type theory* (HoTT) [Uni13]. 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 2.5. 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.

2.2.1.1 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 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 $aa : \mathsf{I} \to A$ 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, I carries operations $\sim, \wedge, \vee$ for manipulating path variables. For instance, the map $\sim : \mathsf{I} \to \mathsf{I}$ 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 $\mathsf{I} \to B$ with definitionally fixed endpoints b_0, b_1, *dependent* paths $aa : \mathsf{PathP}_A\,a_0\,a_1$ are *dependent* functions $(i : \mathsf{I}) \to A\,i$ 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 $(\lambda i \to B : \mathsf{I} \to \mathsf{Type})$. Note that we will use the notation $i.\,A\,i := \lambda i \to A\,i$ to refer to lines of types in inlined code.

2.2.1.2 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 isEquiv can be consulted in [Uni13] 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.

2.2.1.3 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 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 .fst, .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., 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.

2.2.2 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 2.2.1). 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:

$$\frac{\Gamma \vdash i : \mathsf{I} \qquad \Gamma \vdash aa : \mathsf{Path}_A\, a_0\, a_1}{\Gamma \vdash aa\, i : A}$$

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

As can be seen in the second rule, 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 2.2.3. 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 2.2.4.

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 1, 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.

2.2.3 Affine Functions and Bridges

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

2.2.3.1 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 [VV20; VV23]). 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 (2.2.2) is instead expressed as a typechecking side condition denoted $\mathsf{fresh}(f, x)$ (living in the same context Γ as f). We call the latter side condition a freshness constraint and it is defined as the following decidable condition on f, x.

Definition 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 $(v : \mathsf{BI}) \in \Gamma_2$.

[3]Additionally, v can be a variable witnessing the truth of a face constraint (Section 2.5.2) that does not mention x.

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

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 @0 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. 2.1. 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 $\mathsf{red}\, a$. To be more precise about our implementation, some of the rules of Fig. 2.1 contain occurrences of reduction $\mathsf{red}\, a$. 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 2.2.4).

2.2.3.2 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 2.2.3.1. 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.

$$\frac{\Gamma \vdash A : @\mathsf{BI} \to \mathsf{Type} \qquad \Gamma \vdash a_\varepsilon : A\ \mathsf{bi}\varepsilon}{\Gamma \vdash \mathsf{BridgeP}_A\ a_0\ a_1 : \mathsf{Type}}\ \textsc{bdg-f} \qquad \frac{\Gamma \vdash aa_\varepsilon : \mathsf{BridgeP}_A\ a_0\ a_1 \qquad \Gamma, @x : \mathsf{BI} \vdash aa_0\ x = aa_1\ x : A\ x}{\Gamma \vdash aa_0 = aa_1 : \mathsf{BridgeP}_A\ a_0\ a_1}\ \textsc{bdg-}\eta$$

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

$$\frac{\Gamma \vdash aa : \mathsf{BridgeP}_A\ a_0\ a_1 \qquad \Gamma \vdash x : \mathsf{BI} \qquad \mathsf{fresh}(aa, x)}{\Gamma \vdash aa\ x : A\ x}\ \textsc{bdg-e}$$

$$\frac{\Gamma \vdash aa : (@x : \mathsf{BI}) \to A\ x \qquad \Gamma \vdash y : \mathsf{BI} \qquad \mathsf{fresh}(aa, y)}{\Gamma \vdash (\lambda x.\ aa\ x)\ y\ \mapsto\ aa\ y : A\ y}\ \textsc{bdg-}\beta$$

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

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

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

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

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

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

$$\frac{\Gamma \vdash Q : (@x : \mathsf{BI}) \to \mathsf{Gel}\ A_0\ A_1\ R\ x}{\Gamma \vdash \mathsf{ungel}\ \{A_0\}\ \{A_1\}\ \{R\}\ Q : R\ (Q\ \mathsf{bi0})\ (Q\ \mathsf{bi1})}\ \textsc{Gel-e} \qquad \frac{}{\Gamma \vdash \mathsf{Gel}\ A_0\ A_1\ R\ \mathsf{bi}\varepsilon\ \mapsto\ A_\varepsilon : \mathsf{Type}}\ \textsc{Gel-}\partial$$

$$\frac{\Gamma \vdash a_\varepsilon : A_\varepsilon \qquad \Gamma \vdash ar : R\ a_0\ a_1}{\Gamma \vdash \mathsf{ungel}\ (\lambda x.\ \mathsf{gel}\ a_0\ a_1\ ar\ x)\ \mapsto\ ar : R\ a_0\ a_1}\ \textsc{Gel-}\beta \qquad \frac{}{\Gamma \vdash \mathsf{gel}\ a_0\ a_1\ ar\ \mathsf{bi}\varepsilon\ \mapsto\ a_\varepsilon : A_\varepsilon}\ \textsc{gel-}\partial$$

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

Figure 2.1: Rules of the BridgeP, extent and Gel primitives of Agda --bridges. Some premises are omitted.

2.2.4 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. 2.1. 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.

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. 2.1). 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. 2.1). 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 (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[4] $(v : \mathsf{BI}) \in \Gamma_2$. 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 $\pi : \Gamma_1, (x : \mathsf{BI}), \Gamma_2 \to \Gamma_1, (x : \mathsf{BI}), \Upsilon_2$[5]. 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 $\Gamma \vdash \langle x \rangle a : (@y : \mathsf{BI}) \to A\,y$.

2.2.5 Gel Types

The univalence theorem stated in Section 2.2.1.3 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-F in Fig. 2.1).

```
Gel : ∀ (A₀ A₁ : Type) (R : A₀ → A₁ → Type) (@tick x : BI) → 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.

[4] Additionally, v can be a variable witnessing the truth of a face constraint (Section 2.5.2) that does not mention x.

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

2.2.5.1 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-I, GEL-E). 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 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.

2.2.5.2 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 [CH21; CH19] 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 $\sim, \wedge, \vee$ provided by Agda --cubical.

2.2.6 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 $\equiv$ 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.

2.2.7 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 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. 2.2. 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 [CH21]. 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 2.1). Therefore the internal parametricity of Agda --bridges must be used. All calls to parametricity primitives are

```
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
```

Figure 2.2: A low-level proof of a free theorem.

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. 2.2 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 2.3.

2.3 The Observational Parametricity of Agda Bridges

As explained in Section 2.2.7, 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 2.3.1 and illustrated by an example in Section 2.3.1.4.

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 2.3.1. The reason behind this name is that there exists an analogous principle asking instead that all (dependent) types feature a characterization of their $\equiv$/PathP types as types of isomorphisms. The latter principle is known (to varying degrees of generality, see Section 2.6.3) 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 2.3.2. To remedy this situation we introduce in Section 2.3.3 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 2.3.3.1) and (2) derive free theorems for T in a straightforward manner (see the param theorem of Section 2.3.3.2). 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.

2.3.1 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 (2.4) of Section 2.1 in Agda --bridges.

2.3.1.1 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)
```

2.3.1.2 The SRP, Relativistic Reflexive Graphs and the SIP

The structure relatedness principle (SRP) is the following metatheoretical principle:

> **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 2.2.5, and the SRP holds at function types $(a : A) \to B$ thanks to the extentEquiv theorem of Section 2.2.4.

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 $\to$ 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. 2.3.

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 $a_0 : A_0, a_1 : A_1$.

```
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)))
```

Figure 2.3: (Displayed) relativistic reflexive graphs in Agda --bridges, and their →FM semantic rule.

Definition 3 (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 $\eta^\Gamma : \Gamma\{\gamma_0, \gamma_1\} \simeq \mathsf{Bridge}_\Gamma\, \gamma_0\, \gamma_1$.

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 4 (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 $(T,\ \lambda... \to T\{t_0,t_1\}_{\gamma r}\ ,\ \lambda... \to \eta^T)$ is a dRRG over Γ, we allow ourselves to refer to it simply as T. In that case, we also adopt the suggestive notation $(\gamma : \Gamma) \vDash T_\gamma\ \mathsf{dRRG}$ by analogy with the *type* judgment of type theory $(x_0 : A_0, \dots, x_n : A_n \vdash T\ \mathsf{type})$.

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 [AL19] 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 2.6.3 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 2.2.1.3. Types satisfying the SIP are most commonly known as univalent groupoids, or setoids in the literature (c.f. relativistic reflexive graphs).

2.3.1.3 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 2.2.6) 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 2.2.6, 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 (2.2.6) to build the expected equivalence. Our third example regards the List type former. Recall that the triple (Type, $\lambda A_0\, A_1 \to A_0 \to A_1 \to \mathsf{Type}$, 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 2.2.6.

2.3.1.4 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 (2.4) of Section 2.1 in Agda --bridges, or rather a slightly reduced version of it for sparing space.

```
fthm : ∀ {A₀ A₁ : Type} (f : A₀ → A₁) (as₀ : List A₀)
  (p : (X : Type) → List X → List X)  →  map f (p A₀ as₀) ≡ p A₁ (map f as₀)
```

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 2.2.5.1, 2.2.6) 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.

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

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.

```
bare-param {T = λ X → List X → List X → List X} p A₀ A₁ AA
           : BridgeP (λ x → List (AA x) → List (AA x)) (p A₀) (p A₁)
```

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 2.2.7. The proofs are conceptually easier. Moreover they allow for more compositionality. Indeed, contrary to Section 2.2.7 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.

2.3.2 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 1 (The SIP via extensionality principles)**.** By repeatedly applying extensionality principles (see Section 2.2.1.3) we can characterize the meaning of a path between such pointed operations:

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

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 2 (The SRP via relational extensionality principles)**.** Set $\mathsf{Rel}\, X_0\, X_1 := X_0 \to X_1 \to \mathsf{Type}$ and ra := relativity. We can characterize bridges at PointedOp by applying the principles discussed in Section 2.2.5.1, 2.2.6.

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

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 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/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 extentEquiv principle of Section 2.2.4 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 funextEquiv principle of Section 2.2.1.3 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 [Rij22] 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 1 (Fundamental theorem of identity types)**.** *Assume* A : *Type and a relation* Eq : $A \to A \to$ *Type. Suppose that* Eq *is reflexive, i.e.,* $\forall a \to Eq\, a\, a$*. Additionally, suppose that* $\forall a_0 \to isContr\,(\Sigma[\, a_1 \in A\,]\, Eq\, a_0\, a_1)$*. 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 1, 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., $\mathsf{isProp}\, P = \forall p_0\, p_1 \to p_0 \equiv_P p_1$. 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 2 (Heterogenous equality of proofs)**.** *Let* P : $I \to$ *Type be a line of h-propositions, i.e., there is a map isp* : $(i : I) \to isProp\,(P\, i)$*. Suppose that* p_0 : $P\, i0$ *and that* p_1 : $P\, i1$*. Then* $isContr\,(PathP_{i.\, P\, i}\, p_0\, p_1)$[6].

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 $e_0 \equiv e_1 \simeq (e_0.\mathsf{fst} \equiv e_0.\mathsf{fst})$. This is to compare with the relational extensionality principle for equivalences evoked in Section 2.2.5.2 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.

2.3.3 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 4), 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

[6]For bridges, only $\mathsf{isProp}\,(\mathsf{BridgeP}_{x.\, P\, x}\, p_0\, p_1)$ holds. By (2.2.5.1), $\mathsf{BridgeP}_{x.\, \mathsf{Gel}\, P_0\, P_1\, (\lambda - -.\, \bot)\, x}\, p_0\, p_1 \simeq \bot$.

$$\frac{\Gamma\ \mathsf{RRG} \qquad \Gamma \vDash T\ \mathsf{dRRG}}{\Gamma \# T\ \mathsf{RRG}}\ \textsc{ctx-ext} \qquad \frac{\Gamma \vDash A\ \mathsf{dRRG} \qquad \Gamma \# A \vDash B\ \mathsf{dRRG}}{\Gamma \vDash \Pi\mathsf{fm}\ A\ B\ \mathsf{dRRG}}\ \Pi\textsc{fm}$$

$$\frac{\Gamma \vDash A\ \mathsf{dRRG} \qquad \Gamma \# A \vDash B\ \mathsf{dRRG}}{\Gamma \vDash \Sigma\mathsf{fm}\ A\ B\ \mathsf{dRRG}}\ \Sigma\textsc{fm} \qquad \frac{\Gamma \vDash f : \Pi\mathsf{fm}\ A\ B \qquad \Gamma \vDash a : A}{\Gamma \vDash\ \mathsf{app}\ f\ a : B[a]}\ \textsc{app}$$

$$\frac{\Gamma\ \mathsf{RRG}}{\Gamma \vDash \mathsf{Tyfm}\ \mathsf{dNRG}}\ \textsc{tyfm} \qquad \frac{\Gamma \vDash A\ \mathsf{dRRG} \qquad \Gamma \vDash a_0 : A \qquad \Gamma \vDash a_1 : A}{\Gamma \vDash \equiv\mathsf{fm}\ a_0\ a_1\ \mathsf{dRRG}}\ \equiv\textsc{fm}$$

Figure 2.4: Some rules of ROTT. Some premises are not displayed.

in Section 2.3.3.1. As an example, we program in Fig. 2.3 the $\to$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 2.3.3.2.

2.3.3.1 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[7]. 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. 2.4 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. 2.3.

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

[7] In our development those morphisms are called relativistic, or native relators but we will not need this notion here.

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 $\mathsf{Bridge}_{\Gamma \# T}\,(\gamma_0, t_0)\,(\gamma_1, t_1)$, can easily be deduced from the same fact for Γ and T. Regarding $\Pi\mathsf{fm}$ and $\Sigma\mathsf{fm}$, and supposing that Γ is empty, $\Pi\mathsf{fm}\,A\,B$ and $\Sigma\mathsf{fm}\,A\,B$ are RRGs[8] 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 $\mathsf{Type}\{A_0, A_1\} = A_0 \to A_1 \to \mathsf{Type}$ 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 $\mathsf{Bridge}_\Gamma\,\gamma_0\,\gamma_1$, i.e., $\mathsf{bare\text{-}param}\,a\,\gamma_0\,\gamma_1$.

2.3.3.2 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 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.

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

The proof of this theorem is elementary and proceeds in three steps, similar to the proof appearing in Section 2.3.1.4. We omit to write .fst to extract direct maps out of equivalences. First convert the logical relation γr into a bridge $\eta^\Gamma\,\gamma r$: $\mathsf{Bridge}_\Gamma\,\gamma_0\,\gamma_1$. Second use bare parametricity to obtain $\mathsf{bare\text{-}param}\ p\ \gamma_0\ \gamma_1\ (\eta^\Gamma\,\gamma r)$ which has type $\mathsf{Bridge}\mathsf{P}_{x.T(\eta^\Gamma\,\gamma r\,x)}\,(p\,\gamma_0)(p\,\gamma_1)$. 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 ${\eta^T}^{-1}(\mathsf{bare\text{-}param}\ p\ \gamma_0\ \gamma_1\ (\eta^\Gamma\,\gamma r))$ 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. [AKS22] and Section 2.6) 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.

[8] There is a slight mismatch between dRRGs in empty contexts and RRGs, but we ignore it here.

2.4 Internal Observational Parametricity Applied

In this section we obtain several free theorems as one-liner invocations of the param theorem of Section 2.3.3.2. 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.

2.4.1 Reproving fthm and lowChurchBool

We begin by recasting our proof of the global free theorem fthm from Section 2.3.1.4, this time using ROTT and its param rule. Let $p : (X : \mathsf{Type}) \to \mathsf{List}\ X \to \mathsf{List}\ X$, let $f : A_0 \to A_1$ for A_0, A_1 two types and let $as : \mathsf{List}\ A_0$. We want to apply param at program p and at (logical) relation $\mathsf{Gr}\ f : A_0 \to A_1 \to \mathsf{Type}$. 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 $X : \mathsf{Type} \vDash \mathsf{List}\ X \to \mathsf{List}\ X$ dRRG. By the Πfm rule of ROTT (or rather $\to_{\mathrm{FM}}$ of Fig. 2.3) it is sufficient to prove (twice) that $X : \mathsf{Type} \vDash \mathsf{List}\ X$ dRRG. The latter displayed RRG appears as an example in Section 2.3.1.3. 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 2.3.1.4. 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 $(\lambda X \to \mathsf{List}\ X \to \mathsf{List}\ X)\{p\ A_0, p\ A_1\}_{\mathsf{Gr}\ f}$ and contrary to the BridgeP type former, the latter type *reduces* to the expected relational parametricity statement, i.e.:

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

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

Similarly, we can reprove the lowChurchBool theorem of Section 2.2.7. The call to param-prf and the CH-inverse-cond module appearing in Fig. 2.2 are replaced by the following call to the param theorem and definition of an appropriate dRRG using ROTT:

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

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

2.4.2 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 [AAG05]. 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[9]. 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 (2.4). 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. 2.4 the goal is reduced to $X : \mathsf{Type} \vDash F\, X$ dRRG. Recalling that $F\, X = \Sigma[\, s \in S\,]\, P\, s \to X$

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

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 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 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 [ND18].

2.4.3 System F

We can prove Reynolds's abstraction theorem [Rey83] for predicative System F [Lei91] 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 1. *By analogy to Section 2.3.3.2, 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).*

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

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 [NVD17] 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 [CH21] 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. [BCM15]. 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.

2.5 Reimplementing hcomp and transp

Recall from Section 2.2.1 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 $\equiv$) 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 [Coh+17]. 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).

```
transp : (A : I → Type) (φ : I) (u₀ : A i0) → A i1
hcomp : {A : Type} {φ : I} (u : (i : I) → Partial φ A) (u₀ : A) → A
```

We now explain the transp (Section 2.5.1) and hcomp (Section 2.5.2) 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 [VMA21]. This section ends with a brief comparison between the Kan operations implemented by Agda --bridges and those specified by the CH theory.

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

$$\begin{array}{ccc} a_0\,\mathsf{i1} & \dashrightarrow & a_1\,\mathsf{i1} \\ {\scriptstyle a_0}\uparrow & & \uparrow{\scriptstyle a_1} \\ a_0\,\mathsf{i0} & \xrightarrow[u_0]{} & a_1\,\mathsf{i0} \end{array}$$

Setting the result of this reduction to be the bridge $\lambda(x : \mathsf{BI}).\, \mathsf{transp}\,(i.\, A\, i\, x)\, \varphi\, (u_0\, x)$ 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.

2.5.2 Mixed Homogeneous Composition

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

$$\mathsf{mhcomp} : \forall\ \{\mathsf{A} : \mathsf{Type}\}\ \{\zeta : \mathsf{MCstr}\}\ (\mathsf{u} : (\mathsf{i} : \mathsf{I}) \to \mathsf{MPartial}\ \zeta\ \mathsf{A})\ (\mathsf{u}_0 : \mathsf{A}) \to \mathsf{A}$$

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.

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

Similar to transp, the operational semantics of mhcomp is defined by induction on the syntax of its A : 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 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 [Ang+21a; AFH18] (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 ($x = y$: 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 2.2.2) and through semi-freshness in Agda --bridges (see Section 2.2.4).

2.6 Related Work

2.6.1 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 [Wad89] 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 $\mathscr{X}$

Table 2.1: Classification of related work about parametricity, based on [NVD17, 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 $\mathscr{S} = \mathscr{X}$. Ab stands for Agda --bridges. Mt stands for "meta".

Citation	Obj. lang. $\mathscr{X}$	Target lang. $\mathscr{Y}$	$\mathscr{P}$	$\mathscr{S}$	$\mathscr{M}$	mod. in $\mathscr{Y}$ or *(it.)* $\mathscr{M}$
[Rey83]	System F	Mt: Set theory	$\mathscr{Y}$	$\mathscr{Y}$		Sets with relations
[ACC93]	System F	System $\mathscr{R}$	Mt	Mt	Mt	*PERs [BAC95]*
[PA93]	System F	System F + logic	Mt	Mt		
[Wad07]	System F	System F + logic	Mt	Mt		
[Atk12]	System Fω	Mt: Impred. CIC	$\mathscr{Y}$	$\mathscr{Y}$		Reflexive graphs
[Tak01]	$\mathscr{X} \in \lambda$-cube	$\mathscr{X}+\forall+\Pi \in \lambda$-cube	Mt	Mt		
[BJP12]	Any PTS	Suitable PTS	Mt	Mt		
[TTS21]	CIC	Univalent CIC	Mt	Mt		
[KD13]	CC	Mt	Mt	Mt		PERs
[AGJ14]	MLTT	Mt: CIC	$\mathscr{Y}$	$\mathscr{Y}$		Reflexive graphs
[BM12]	BM	$\mathscr{X}$	$\mathscr{X}$	$\mathscr{X}$		*none*
[BCM15]	BCM	$\mathscr{X}$	$\mathscr{X}$	Mt	Mt	*Affine cubical sets*
[NVD17]	ParamDTT	$\mathscr{X}$	$\mathscr{X}$	Mt	Mt	*Bdg/path cub. sets*
[ND18]	RelDTT	$\mathscr{X}$	$\mathscr{X}$	Mt	Mt	*Depth n cub. sets*
[CH21]	CH	$\mathscr{X}$	$\mathscr{X}$	Mt	Mt	*Aff. /cart. bicub. sets*
Ab	Ab	$\mathscr{X}$	$\mathscr{X}$	Mt	Mt	*Aff. /CCHM bicub. sets*
ROTT	DTT ($\rightsquigarrow$ Ab)	Ab	$\mathscr{Y}$	$\mathscr{Y}$		RRGs
Corr. 1	Pr. Sys. F	Ab	$\mathscr{Y}$	$\mathscr{Y}$		RRGs
[Alt+24]	ACKS	$\mathscr{X}$	$\mathscr{X}$	$\mathscr{X}$	Mt	*Affine cubical sets*
C. ROTT	$\approx$ ACKS	$\mathscr{X}$	$\mathscr{X}$	$\mathscr{X}$	Ab	*Relativistic cub. sets*

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 2.3.1.2) 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 2.1 (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 [Rey84]. 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.

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 [BM12] 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 2.4.3). We refer to Nuyts [Nuy21] 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. [AGJ14], 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 (see C. ROTT entry in the table). 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. [Alt+24], who present an internally parametric type theory which avoids the use of an interval and validates the SRP definitionally.

2.6.2 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 [TTS21] work in the previous section. A reformulation of their work using our techniques would effectively lead to a shallow embedding of observational setoid [PT22] or homotopy [AKS22] type theory in a CwF of univalent setoids or groupoids.

Awodey et al. [AFS18] 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.

2.6.3 The Structure Identity Principle (SIP)

We discuss appearances of the SIP in the literature and compare these to our discussion in Section 2.3. 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 [Uni13] defines a *notion of structure* on a category $\mathcal{C}$ essentially as a displayed category $\mathcal{D}^\bullet$ over $\mathcal{C}$ such that the projection functor $P : \Sigma\mathcal{C}\mathcal{D}^\bullet \to \mathcal{C}$ from the total space is faithful, implying that the fiber of any object of $\mathcal{C}$ (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 $\mathcal{C}$ is univalent and $\mathcal{D}^\bullet$ is standard, then the total space $\Sigma\mathcal{C}\mathcal{D}^\bullet$ 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 $\Sigma\mathcal{C}\mathcal{D}^\bullet$, 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. [Ang+21b] 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 2), 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. [Ahr+20, thm. 7.10] use FOLDS [Mak95] 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.

Chapter 3

Nominal Type Theory by Nullary Internal Parametricity

This chapter is an exact reproduction, up to minor cosmetic modifications, of a peer-reviewed paper (with the same title) accepted for publication in the post-proceedings of the TYPES25 conference [VND25].

The research presented in this chapter was carried out by me, with supervision and contributions from Dominique Devriese and Andreas Nuyts. They helped shape the ideas presented in this chapter and contributed to the writing of the text. In particular Section 3.3 was mainly written by Andreas Nuyts.

3.1 Introduction

There are many ways to formally define the syntax of a language with binders. In particular, nominal frameworks [Pit03; SPG03; PMD14; SS04; Che12; Urb08] are metalanguages featuring (among others) a primitive type of names Nm as well as a "name abstraction" type former which we write @N ⊸ −. Name abstraction types can be used to specify the type of binders of a given object language. For example, we can define the syntax of the untyped lambda calculus (ULC) with the following data type (we use syntax similar to Agda, eventhough Agda does not feature a type of names and the name abstraction type former).

```
data Ltm : 𝒰 where
  var : Nm → Ltm
  app : Ltm → Ltm → Ltm
  lam : (@N ⊸ Ltm) → Ltm
```

The resulting representation is called a "nominal data type" or "nominal syntax"

and offers several advantages. First, α-equivalent terms are definitionally equal, a consequence of how $@\mathsf{N} \multimap -$ is axiomatized. Second, nominal data types have a (closed) induction principle. Third, by contrast with the De Bruijn representation, weakening by a name simply means extending the context with that name, and all programs in the metalanguage are automatically stable under this operation. Fourth, formal reasoning with nominal syntax can sometimes match on-paper reasoning [Urb08]. A drawback of this representation is that it is not definable in plain type theory.

Inference rules for the name abstraction type former can be defined in two ways: either in an existential/positive, or in a universal/negative fashion. Interestingly, these presentations are semantically equivalent [SS04; Sch06] but syntactically have different pros and cons.

On the one hand, the existential presentation makes the name abstraction type former behave as an existential quantification on names. So an element of that type, i.e. a name abstraction, can be understood as a *name-term pair up to α-renaming* $\langle x, a\rangle$ where a may bind the freshly chosen name x. This presentation is convenient since the user is allowed to *pattern match* on such "binding" name-term pairs, an ability that we call nominal pattern matching. The following example is Example 2.1 from [SPG03] and illustrates this ability. It is a (pseudo-) FreshML program computing the equality modulo α-renaming of two existential name abstractions. In the example A has decidable equality $\mathsf{eq}_A : A \to A \to \mathsf{Bool}$ and the existential name abstraction type is written $@\mathsf{N} \cdot -$.

```
eqabs : (@N · A) → (@N · A) → Bool
eqabs ⟨x₀,a₀⟩ ⟨x₁,a₁⟩ = eq_A (swap x₀,x₁ in a₀) a₁
```

Note (1) the occurrences of x_0, x_1 outside of $\langle x_0, a_0\rangle, \langle x_1, a_1\rangle$ and (2) the appearance of the swap operation exchanging free names.

Nominal pattern matching is convenient and for that reason nominal frameworks use the existential presentation for practical reasoning about nominal syntax. However, inference rules for existential abstraction types are cumbersome to specify. The issue is that a name x is considered fresh in a binding pair $\langle x, a\rangle$, and such information must be encoded at the type level and propagated when pattern matching. The consequence is that the rules for existential abstraction types are polluted with typal freshness information. This makes the implementation of such rules harder in a proof assistant environment.

On the other hand, the universal presentation treats name abstractions not as pairs, but rather as *functions* consuming fresh names (a.k.a. affine or fresh functions), as in [PMD14; Che12]. This has several consequences: names can only be used when they are in scope, no explicit swapping primitive is needed and the inference rules are straightforward to specify. However one seems to lose the important ability to pattern match.

In this paper we will propose a new foundation for nominal frameworks alleviating

the above issues and relying on the notion of *parametricity*. Concisely put, n-ary parametricity asserts that types behave as n-ary reflexive graphs. More precisely, the n-ary parametricity of a type T computes by induction on T a certain n-ary reflexive graph structure for T. Even more precisely, (1) the n-ary parametricity of a (open) type T computes to a certain (open) n-ary relation written $[T]_n : T \to \ldots \to T \to \mathcal{U}$, (2) the n-ary parametricity of a (open) term $t : T$ computes to a certain reflexivity edge at t, i.e. $[t]_n : [T]_n\, t\, \ldots\, t$. Parametricity is most often used in its binary form, at types of polymorphic functions, where it specializes to a uniformity property [Rey83; Wad89]. For example if $t : T = \forall(X : \mathcal{U}).\, X \to X$ then $[t]_2 : [T]_2\, t\, t$ is a proof that t maps related types to related functions. Indeed $[-]_2$ is defined by induction (not given here) and $[T]_2\, t\, t$ reduces to $\forall(X_0 X_1 : \mathcal{U})(R : X_0 \to X_1 \to \mathcal{U}).\forall x_0\, x_1.\, R\, x_0\, x_1 \to R\, (t\, X_0\, x_0)\, (t\, X_1\, x_1)$. Intuitively this property disallows t from inspecting its type argument $X : \mathcal{U}$. It can be shown that the property entails that t equals the identity $\lambda\, X\, x.\, x$.

In the above account, parametricity is regarded as a property that is defined and proven to hold *externally* about a dependent type theory (DTT), as in [BJP12]. In other words, the parametricity property, or translation, is a map $[-]_n$ defined by induction on the (open) types and terms of DTT. By contrast, n-ary *internally* parametric type theory (n-ary PTT for short) [CH21; Mou16; ND18; NVD17; Alt+24; Abe24] extends DTT with new type and term formers. These primitives make it possible to prove parametricity results *within* n-ary PTT.

To that end, n-ary PTT typically provides two kinds of primitives internalizing aspects of reflexive graphs. Firstly, Bridge types are provided, which intuitively are to a type what an edge is to a reflexive graph. Syntactically, a bridge q : $\mathsf{Bridge}\, A\, a_0\, \ldots\, a_{n-1}$ at type A between $a_0, \ldots, a_{n-1}$ is treated as a function $\mathsf{N} \to A$ out of a posited bridge interval N. The N interval contains n endpoints $(e_i)_{i<n}$ and q must respect these definitionally $q\, e_i = a_i$. When $n = 2$ this is similar to the "paths" of cubical type theory [VMA21] which play the role of equality proofs in the latter. However, bridges do not satisfy various properties of paths, e.g. they cannot be composed and one cannot transport values of a type $P\, x$ over a bridge $\mathsf{Bridge}\, A\, x\, y$ to type $P\, y$. Moreover, contrary to paths, bridges may only be applied to variables $x : \mathsf{N}$ that do not appear freely in them, i.e. that are fresh for the bridge. Functions with such a freshness side condition are also known as affine or fresh functions and we will use the symbol $\multimap$ instead of $\to$ for their type. Secondly, the other primitives of n-ary PTT make it possible to prove that the Bridge type former has a commutation law with respect to every other type former. Table 3.1 lists some of these laws (equivalences) for arity $n = 2$ and compares the situation for bridges and paths. Path types are written $\equiv$.

The last column in Table 3.1 excerpts a collection of equivalences known as the Structure Identity Principle (SIP) in HoTT/UF [Uni13]. Concisely, it states that at every type K, equality is equivalent to observational equality [AMS07]. The Bridge column of Table 3.1 shows an analogous Structure Relatedness Principle (SRP) [VND24] which

$K : \mathcal{U}$	$\mathsf{Bridge}\, K\, k_0\, k_1 \simeq \ldots$	$k_0 \equiv_K k_1 \simeq \ldots$
$A \to B$	$\forall a_0\, a_1.\ \mathsf{Bridge}\, A\, a_0\, a_1 \to \mathsf{Bridge}\, B\, (k_0\, a_0)\, (k_1\, a_1)$	$\forall\, a_0\, a_1.\, a_0 \equiv_A a_1 \to k_0\, a_0 \equiv_B k_1\, a_1$
$A \times B$	$\mathsf{Bridge}\, A\, (k_0\, .\mathsf{fst})(k_1\, .\mathsf{fst}) \times \mathsf{Bridge}\, B\, (k_0\, .\mathsf{snd})(k_1\, .\mathsf{snd})$	$(k_0\, .\mathsf{fst}) \equiv_A (k_1\, .\mathsf{fst}) \times (k_0\, .\mathsf{snd}) \equiv_B (k_1\, .\mathsf{snd})$
$\mathcal{U}$	$k_0 \to k_1 \to \mathcal{U}$	$k_0 \simeq k_1$

Table 3.1: The Bridge and Path type formers commute with some example type formers.

expresses equivalence of K's Bridge type to the parametricity translation of K, i.e. the Bridge type former internalizes the parametricity translation for types *up to SRP equivalences*. Accordingly, in this setting the parametricity of a term k is k's reflexivity bridge $\lambda(\_ : \mathsf{N}).\, k : \mathsf{Bridge}\, K\, k\ \ldots\ k$ mapped through the SRP equivalence at K.

Parametric Nominal Type Theory In this work, we put forward and elaborate an idea that existed in the community [Nar25b; ND24], which is to use nullary PTT (i.e. 0-ary PTT) as the foundation for nominal dependent type theory. Specifically, we contribute Parametric Nominal Type Theory (PNTT). Concisely put, PNTT = nullary PTT + a type of names with an induction principle + rules for nominal data types.

In more detail, PNTT is obtained from nullary PTT in several steps. First, we make the simple observation that universal name abstraction types have the same rules as the nullary Bridge types from nullary PTT: nullary bridges and universal abstractions are simply affine functions out of the interval. Second, we show that, on its own, nullary PTT already supports important features of nominal frameworks: universal name abstractions (bridge types), typal freshness, (non-primitive) existential name abstractions, a name swapping operation and the local-scoping primitive ν of [PMD14] (roughly, the latter primitive is used to witness freshness). Third, we can recover nominal pattern matching in this parametric setting by extending nullary PTT with a full-fledged type of names $\vdash \mathsf{Nm}$. It comes equipped with a novel eliminator called *name induction*, which expresses for a term $\Gamma \vdash n : \mathsf{Nm}$ and a bridge variable x in Γ, that either n is just x, or x is *fresh* in n. Fourth, PNTT also has rules for the nominal data types we wish to study, like the Ltm type.

Nominal pattern matching can be indirectly recovered via an extended version of the nullary SRP: the Structure Abstraction Principle (SAP), as we call it. The SAP expresses that the nullary bridge type former $@\mathsf{N} \multimap -$ commutes with every type former in a specific way, including (1) standard type formers, (2) the Nm type and (3) nominal data types. For standard type formers, the SAP asserts that name abstraction commutes as one might expect, e.g. $(@\mathsf{N} \multimap A \to B) \simeq ((@\mathsf{N} \multimap A) \to (@\mathsf{N} \multimap B))$. For Nm, the SAP asserts that $(@\mathsf{N} \multimap \mathsf{Nm}) \simeq 1 + \mathsf{Nm}$, which can be proved by name induction. The SAP also holds for nominal data types. For example, the SAP for Ltm asserts that there is an equivalence $e : (@\mathsf{N} \multimap \mathsf{Ltm}) \simeq \mathsf{Ltm}_1$

between $@\mathsf{N} \multimap \mathsf{Ltm}$ and the following data type (this was proved semantically by Hofmann [Hof99]). Intuitively, it corresponds to Ltm terms with Nm-shaped holes and we say it ought to be the nullary parametricity translation of Ltm.

```
data Ltm₁ : 𝒰 where
  hole : Ltm₁
  var  : Nm → Ltm₁
  app  : Ltm₁ → Ltm₁ → Ltm₁
  lam  : (@N ⊸ Ltm₁) → Ltm₁
```

When matching on $\mathsf{lam}\, g : \mathsf{Ltm}$ we have $g : @\mathsf{N} \multimap \mathsf{Ltm}$ and equivalently $e\, g : \mathsf{Ltm}_1$ for which the induction principle of Ltm_1 applies.

Contributions and Outline In this paper, we propose nullary PTT as the foundation for nominal dependent type theory, and provide evidence for the suitability of this foundation.

- In Section 3.2, we present Parametric Nominal Type Theory (PNTT). PNTT is a variant of the univalent parametric type theory of Cavallo and Harper [CH21] where (1) we replace the arity $n = 2$ by $n = 0$ (see Section 3.2.1), (2) we construct a type Nm from the bridge interval @N and provide a name induction principle (see Section 3.2.2), (3) we add rules for the specific nominal data types we wish to study. We explain how the above primitives validate our Structure Abstraction Principle (SAP). We discuss semantics and soundness in Section 3.2.4.
- In Section 3.3, we systematically discuss the inference rules of existing nominal frameworks for typal freshness, name swapping, the local-scoping primitive ν, existential and universal name abstractions. We explain in detail how they can all be (adapted and) implemented in terms of nullary PTT primitives.
- Section 3.4 shows nominal techniques in PNTT in action. Section 3.4.1 shows that the nullary translation of data types lets us indirectly emulate nominal pattern matching: defining functions by matching on patterns which bind variables. Section 3.4.2 connects the Ltm nominal data type to a nominal HOAS representation, and serves two purposes. First, it illustrates that more often than not, proving the correctness of a function $f : D \to E$ defined by recursion out of a nominal data type D requires computing the parametricity translation of f, i.e. the parametricity of f is term-relevant. Second, the example sheds some light on the notion of *Kripke* binary parametricity [Atk09]. A proof in Atkey's Kripke model is ported to PNTT, and the Kripke aspect of it is emulated by nullary parametricity.

Binary parametric cubical type theory has already been implemented [VND24]. Thus our work offers a clear path to a practical implementation of nominal type theory. To our knowledge, we are also the first to make explicit and active use of nullary parametricity. We discuss related work in more detail in Section 3.5.

3.2 Parametric Nominal Type Theory (PNTT)

PNTT is largely based on the binary PTT of Cavallo and Harper (CH type theory, [CH21]). Concretely, our rules are obtained by (1) considering the rules of the parametricity primitives of the CH binary PTT and replacing the arity 2 by 0 instead, (2) adding novel rules to turn the bridge interval $@\mathsf{N}$ into a full-fledged type Nm, including a name induction principle and (3) adding rules for the desired nominal data types. For the impatient reader, the relevant rules of PNTT appear in Fig. 3.1 and $\mathsf{Gel}\, A\, x$ can be understood as the elements of A for which x is fresh.

Cubical type theory Apart from its parametricity primitives, the CH theory is a cubical type theory in the way it handles equality proofs (*paths*), and so is our system. Cubical type theory (cubical TT, [Coh+17; Ang+21a]) is a form of homotopy type theory (HoTT, [Uni13]) that adds new types, terms and equations on top of plain dependent type theory. These extra primitives make it possible, among other things, to prove univalence and more generally the SIP (see Table 3.1) at any concrete, fully known type.

The SIP matters to us because, e.g., univalence is used to characterize the bridge type at the universe $(@\mathsf{N} \multimap \mathcal{U}) \simeq \mathcal{U}$. Otherwise, the fact that our system is a cubical type theory is not of primary significance, although we wish to remain as close as possible to the CH theory to be able to reuse their proofs and semantics.

Accordingly, we only need to know that the primitives introduced by cubical type theory validate the SIP in order to showcase our theorems and examples. Yet we list these primitives for completeness: (1) the path interval I, dependent path types and their rules; paths play the role of equality proofs thus non-dependent path types are written $\equiv$, (2) the transport and cube-composition operations, a.k.a the Kan operations; these are used e.g. to prove transitivity of the path relation, (3) a type former to turn equivalences into paths in the universe, validating one direction of univalence, (4) higher inductive types (HITs) if desired.

Additionally, in HoTT an equivalence $A \simeq B$ is by definition a function $A \to B$ with contractible fibers. We rather build equivalences using Theorem 4.2.3 of [Uni13]: let $f : A \to B$ have a quasi-inverse, i.e. a map $g : B \to A$ satisfying the two roundtrip equalities $\forall a.\, g(f\, a) \equiv a$ and $\forall b.\, f(g\, b) \equiv b$. Then f can be turned into an equivalence $A \simeq B$.

3.2.1 Nullary CH

Let us explain the rules of Fig. 3.1. We begin with the nullary primitives of the CH type theory since our rules for the bridge interval Nm depend on them.

Contexts There are two ways to extend a context. The first way is the usual "cartesian" comprehension operation where Γ gets extended with a type $\Gamma \vdash A\,\mathsf{type}$

$$\frac{\Gamma, (x : @\mathsf{N}) \vdash A\ \mathsf{type}}{\Gamma \vdash (x : @\mathsf{N}) \multimap A\ \mathsf{type}}\ {\multimap}\mathrm{F} \qquad \frac{\Gamma, (x : @\mathsf{N}) \vdash a : A}{\Gamma \vdash \lambda\, x.\, a : (x : @\mathsf{N}) \multimap A}\ {\multimap}\mathrm{I}$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x \vdash a' : (y : @\mathsf{N}) \multimap A}{\Gamma \vdash a'\, x : A[x/y]}\ {\multimap}\mathrm{E} \qquad \frac{(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x, (y : @\mathsf{N}) \vdash a : A}{\Gamma \vdash (\lambda\, y.\, a)\, x = a[x/y] : A[x/y]}\ {\multimap}\beta$$

$$\frac{\Gamma, (x : @\mathsf{N}) \vdash a'_0\, x = a'_1\, x : A}{\Gamma \vdash a'_0 = a'_1 : (x : @\mathsf{N}) \multimap A}\ {\multimap}\eta$$

$$\frac{\begin{array}{c}(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x, (y : @\mathsf{N}) \vdash A\ \mathsf{type} \quad \Gamma\backslash x, (y : @\mathsf{N}), (a : A\, y) \vdash B\ \mathsf{type} \\ \Gamma\backslash x, (a' : (z : @\mathsf{N}) \multimap A[z/y]), (y : @\mathsf{N}) \vdash b : B[a'\, y/a] \qquad \Gamma \vdash a_x : A[x/y]\end{array}}{\Gamma \vdash \mathsf{ext}\ (\lambda\, a'\, y.\, b)\, x\, a_x : B[x/y, a_x/a]}\ \textsc{ext}$$

$$\frac{\text{prem. of } \textsc{ext} - a_x \quad \Gamma\backslash x, (y : @\mathsf{N}) \vdash a : A}{\Gamma \vdash \mathsf{ext}\ (\lambda\, a'\, y.\, b)\ x\ (a[x/y]) = b[\lambda\, y.\, a/a', x/y] : B[x/y, a[x/y]/a]}\ \textsc{ext}\beta$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x \vdash A\ \mathsf{type}}{\Gamma \vdash \mathsf{Gel}\, A\, x\ \mathsf{type}}\ \textsc{Gel}\mathrm{F} \qquad \frac{\Gamma, (x : @\mathsf{N}) \vdash g : \mathsf{Gel}\, A\, x}{\Gamma \vdash \mathsf{ung}\ (\lambda\, x.\, g) : A}\ \textsc{Gel}\mathrm{E}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{ung}\ (\lambda\, x.\, \mathsf{gel}\, a\, x) = a}\ \textsc{Gel}\beta$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x \vdash a : A}{\Gamma \vdash \mathsf{gel}\, a\, x : \mathsf{Gel}\, a\, x}\ \textsc{Gel}\mathrm{I} \qquad \frac{\begin{array}{c}\text{prem. of } \textsc{Gel}\mathrm{F} \quad \Gamma\backslash x, (y : @\mathsf{N}) \vdash g_0, g_1 : \mathsf{Gel}\, A\, y \\ \Gamma \vdash \mathsf{ung}(\lambda y.\, g_0)\ =\ \mathsf{ung}(\lambda y.\, g_1) : A\end{array}}{\Gamma \vdash g_0[x/y] = g_1[x/y] : \mathsf{Gel}\, A\, x}\ \textsc{Gel}\eta$$

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Nm}\ \mathsf{type}}\ \mathsf{Nm}\mathrm{F} \qquad \frac{(x : @\mathsf{N}) \in \Gamma}{\Gamma \vdash \mathsf{c}\, x : \mathsf{Nm}}\ \mathsf{Nm}\mathrm{I}$$

$$\frac{\begin{array}{c}(x : @\mathsf{N}) \in \Gamma \quad \Gamma \vdash n : \mathsf{Nm} \quad \Gamma, z : \mathsf{Nm} \vdash B\ \mathsf{type} \quad \Gamma \vdash b_0 : B[\mathsf{c}\, x/z] \\ \Gamma, (g : \mathsf{Gel}\, \mathsf{Nm}\, x) \vdash \mathsf{forg}\, x\, g := \mathsf{ext}\ (\lambda\, g'\, y.\, \mathsf{ung}\, g')\, x\, g : \mathsf{Nm} \\ \Gamma, (g : \mathsf{Gel}\, \mathsf{Nm}\, x) \vdash b_1 : B[\mathsf{forg}\, x\, g/z]\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, n\, b_0\ (\lambda\, g.\, b_1) : B[n/z]}\ \mathsf{Nm}\mathrm{E}$$

$$\frac{\text{prem. of } \mathsf{Nm}\mathrm{E} - n}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, (\mathsf{c}\, x)\, b_0\ (\lambda\, g.\, b_1) = b_0 : B[\mathsf{c}\, x/y]}\ \mathsf{Nm}\beta_0$$

$$\frac{\text{prem. of } \mathsf{Nm}\mathrm{E} - n \quad \Gamma\backslash x \vdash n : \mathsf{Nm}}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, n\, b_0\ (\lambda\, g.\, b_1) = b_1[\mathsf{gel}\, n\, x/g] : B[n/z]}\ \mathsf{Nm}\beta_1$$

Figure 3.1: Parametricity primitives of PNTT. Prem. of x - y means premises of rule x except y. For binding rules such as EXT, we rely on the invertibility of both variable and name abstraction to bind using λ.

resulting in a context $\Gamma, (a : A)$. The second, distinct way to extend a context Γ is an "affine" comprehension operation where Γ gets extended with a bridge variable x resulting in a context written $\Gamma, (x : @\mathsf{N})$. Note that $@\mathsf{N}$ is *not* a type and morally always appears on the left of $\vdash$ (the Bridge type former will be written $@\mathsf{N} \multimap -$ but this is just a suggestive notation).

The presence of $@\mathsf{N}$ is required because the theory treats affine variables in a special way that is ultimately used to prove the SAP (briefly mentioned in Section 3.1). Specifically, terms, types and substitutions depending on an affine variable $x : @\mathsf{N}$ are not allowed to duplicate x in affine positions. So typechecking an expression that depends on $x : @\mathsf{N}$ may involve checking that x does not appear freely in some subexpressions. Since variables declared after x in the context may eventually be substituted by terms mentioning x, a similar verification must be performed for them. More formally, such "freshness" statements about free variables are specified in the inference rules using a context restriction operation $-\backslash-$ (as in [CH21], section 2.1). If Γ is a context containing $(x : @\mathsf{N})$ then $\Gamma\backslash x$ is the context obtained from Γ by removing x itself, as well as all the cartesian (i.e. not $@\mathsf{N}$) variables to the right of x.[1] If $(x : @\mathsf{N}) \in \Gamma$ and $\Gamma\backslash x \vdash a : A$ we say that x is fresh in a.

Nullary bridges The type former of dependent bridges with dependent codomain $\Gamma, (x : @\mathsf{N}) \vdash A$ is written $(x : @\mathsf{N}) \multimap A$. The type of non-dependent bridges with codomain $\Gamma \vdash A$ is written $@\mathsf{N} \multimap A$ and defined as $(\_ : @\mathsf{N}) \multimap A$. Introducing a bridge requires providing a term in a context extended with a bridge variable $(x : @\mathsf{N})$. A bridge $a' : (y : @\mathsf{N}) \multimap A$ can be eliminated at a variable $(x : @\mathsf{N})$ only if x is fresh in a'. In summary, nullary bridges are treated and written like (dependent) functions out of the $@\mathsf{N}$ pretype but are restricted to consume fresh variables only. From a nominal point of view, a bridge $a' : @\mathsf{N} \multimap A$ is a name-abstracted value in A.

Since we will use the theory on non-trivial examples in the next sections, we prefer to explain the other primitives of Fig. 3.1 from a user perspective: we derive programs corresponding to these primitives in the empty context and express types as elements of the universe type $\mathcal{U}$, whose rules are not listed but standard. Furthermore we explain what the equations of Fig. 3.1 entail for these closed programs. The closed programs derived from the CH nullary primitives have a binary counterpart in [VND24], an implementation of the CH binary PTT. The nullary and binary variants operate in a similar way. The pseudo-code we write uses syntax similar to Agda. We omit universe levels, and $\multimap$ is parsed like $\to$, e.g., it is right associative. We use ; to group several declarations in one line.

Lastly we indicate that the SAP holds at equality types $(a'_0\, a'_1 : @\mathsf{N} \multimap A) \to ((x : @\mathsf{N}) \multimap a'_0 x \equiv_A a'_1 x) \simeq a'_0 \equiv a'_1$. Proofs of this fact in the binary case can be found in [CH21; VND24]. The nullary proof is obtained by erasing all mentions of (bridge) endpoints.

The extent primitive The rule for the extent primitive (EXT) together with the rules of $\multimap$ and $\mathcal{U}$ provide a term ext in the empty context with the following type.

[1] A detail: variables $i : \mathsf{I}$ and cubical constraints are not removed [CH21; VND24] as they cannot depend on x.

```
ext : {A : @N ⊸ 𝒰} {B : (x : @N) ⊸ A x → 𝒰}
  (f' : (a' : (x : @N) ⊸ A x) → (x : @N) ⊸ B x (a' x)) →
  (x : @N) ⊸ (a : A x) → B x a
```

From a function f' mapping a bridge in A to a bridge in B, the extent primitive lets us build a bridge in the dependent function type formed from A and B, or $\Pi\, A\, B$ for short. Concisely put, the extent primitive validates one direction of the SAP at Π. It can be upgraded into the following equivalence, using the β-rule of extent, described below (as a side note, the proof also uses a lemma akin to a propositional η-rule for extent).

$$\begin{aligned}&((a' : (x : @\mathsf{N}) \multimap A\, x) \to (x : @\mathsf{N}) \multimap B\, x\, (a'\, x))\\ &\quad\simeq (x : @\mathsf{N}) \multimap (a : A\, x) \to B\, x\, a\end{aligned}$$

By way of comparison, this equivalence appears in [PMD14] (Example 2.2) though the left-to-right direction is implemented via a composite capturing operation instead of a primitive like extent. Proofs of the above equivalence in the binary case can be found in [CH21; VND24]. The nullary proof is obtained by erasing all mentions of endpoints. This equivalence entails $((@\mathsf{N} \multimap A) \to (@\mathsf{N} \multimap B)) \simeq @\mathsf{N} \multimap (A \to B)$ in the non-dependent case.

The β-rule of extent (EXTβ) is peculiar because a substituted premise $a[x/y]$ appears in the redex on the left-hand side. To compute the right-hand side out of the left-hand side only, the premise a is rebuilt (*) and a term where a variable is bound in a is returned. The step (*) is possible thanks to the restriction in the context $\Gamma\backslash x, (y : @\mathsf{N})$ of a. This is quite technical and explained in [CH21; VND24]. We instead explain what that entails for our ext program above. Non-trivial examples of β-reductions for extent will appear in Section 3.2.2.

In a context Γ containing $(x : @\mathsf{N})$, the term $\Gamma \vdash \mathsf{ext}\ f'\ x\ a$ reduces if $\Gamma \vdash a$ is a term that does not mention cartesian variables strictly to the right of x in Γ, i.e. declared after x in Γ. In particular a can mention x. If such a freshness condition holds, x can in fact be soundly captured in a and the reduction can trigger. By default the reduction does not trigger because $f', x, a \vdash \mathsf{ext}\ f'\ x\ a$ does not satisfy the freshness condition as in this case a is a variable appearing to the right of x in the context.

Gel types Internally parametric type theories that feature interval-based Bridge types [Mou16; CH21] need a primitive to convert an n-ary relation of types into an n-ary bridge. In the CH theory the primitive is called Gel. The rules for Gel types together with the rules of $\multimap$ and $\mathcal{U}$ imply the existence of the following programs in the empty context. Note that ung is called ungel in the CH theory.

```
Gel : 𝒰 → @N ⊸ 𝒰
gel : {A : 𝒰} → A → (x : @N) ⊸ Gel A x
ung : {A : 𝒰} → ((x : @N) ⊸ Gel A x) → A
```

Similar to extent, Gel validates one direction of the SAP, this time at the universe $\mathcal{U}$. The Gel function can be upgraded into an equivalence $\mathcal{U} \simeq (@\mathsf{N} \multimap \mathcal{U})$ where $\mathsf{Gel}^{-1}\, A' := (x : @\mathsf{N}) \multimap A'\, x$, and by using the rules of Gel and ext, and notably univalence. The proof also requires showing that gel and ung are inverses, i.e. the SAP at Gel types $A \simeq (x : @\mathsf{N}) \multimap \mathsf{Gel}\, A\, x$. Again these theorems are proved in [CH21; VND24] in the binary case, and the nullary proofs are obtained by erasing endpoints.

From a nominal perspective, we observe that Gel is a type former that can be used to express freshness information. This can be seen by looking at the GelI rule: canonical inhabitants of Gel $A\, x$ are equivalently terms $a : A$ such that x is fresh in a. Conversely the ung primitive is a binder that makes available a fresh variable x when a value of type A is being defined, as long as the result value is typally fresh w.r.t. x. The ung primitive can also be used to define a map forgetting freshness information.

```
forg : {A : 𝒰} → (x : @𝟙) ⊸ Gel A x → A
forg {A} = ext [λ (g' : (x : @𝟙) ⊸ Gel A x). λ (_ : @𝟙). ung g']
```

To our knowledge the equivalence $\mathcal{U} \simeq (@\mathsf{N} \multimap \mathcal{U})$ had not been considered in the literature on nominal techniques. Lastly, Gelη can be used if a certain freshness side condition holds, similar to extβ. We will not need to make explicit use of Gelη.

3.2.2 The Nm type, Name Induction, Nominal Data Types

So far the rules we presented involved occurrences of the bridge interval morally to the left of $\vdash$, as affine comprehensions. Since we wish to use nullary PTT as a nominal framework, we need a first-class type of names $\vdash$ Nm type. Indeed this is needed to express nominal data types, whose constructors may take names as arguments. For example the var constructor of the Ltm type declared in Section 3.1 has type var : Nm → Ltm, a regular "cartesian" function type. Constructors must be cartesian functions because that is what the initial algebra semantics of (even nominal) data types dictates. Besides, it is unclear whether there exists a sound notion of data types D with "constructors" of the form $@\mathsf{N} \multimap D$.

Perhaps the most important result in our system is the SAP at the type of names Nm which reads $1 + \mathsf{Nm} \simeq (@\mathsf{N} \multimap \mathsf{Nm})$. The principle expresses that a bridge at the type of names Nm is either the "identity" bridge c : @𝟙 ⊸ Nm or a constant bridge. This principle is proved based on our rules for Nm, explained now. The rules of Nm together with the other rules of Fig. 3.1 entail the existence of the following programs in the empty context.

```
Nm : 𝒰
c : @𝟙 ⊸ Nm
indNm : {B : Nm → 𝒰} →
  (x : @𝟙) ⊸ (n : Nm) →
  (b0 : B (c x)) → (b1 : (g : Gel Nm x) → B (forg x g)) → B n
```

The NmI rule expresses that the canonical inhabitants of Nm in a context Γ are simply the affine variables $(x : @\mathsf{N})$ appearing in Γ. The "identity" bridge program c above is derived from that rule. In simple terms, c coerces affine bridge variables x into names $\mathsf{c}\,x : \mathsf{Nm}$.

Name induction The $\mathsf{ind}_{\mathsf{Nm}}$ program/rule is an induction principle, or dependent eliminator for the type Nm. It expresses that in a context Γ containing an affine variable $(x : @\mathsf{N})$ we can do a case analysis on a term $\Gamma \vdash n : \mathsf{Nm}$. Either x is bound in n and n is in fact exactly $\mathsf{c}\,x$, or x is *fresh* in n. The call $\mathsf{ind}_{\mathsf{Nm}}\,x\,n\,b_0\,b_1$ returns b_0 if $n = \mathsf{c}\,x$ (see rule $\mathsf{Nm}\beta_0$) and returns $b_1\,(\mathsf{gel}\,n\,x)$ if x is fresh in n (see rule $\mathsf{Nm}\beta_1$). The freshness assumption in b_1 is expressed typally using a Gel type.

We show that the rules $\mathsf{Nm}\beta_0$ and $\mathsf{Nm}\beta_1$ are well-typed, i.e. that the $\mathsf{ind}_{\mathsf{Nm}}$ program above reduces to something of type $B\,n$ in both scenarios. If $n = \mathsf{c}\,x$ then the reduction result is $\mathsf{ind}_{\mathsf{Nm}}\,x\,(\mathsf{c}\,x)\,b_0\,b_1 = b_0 : B\,(\mathsf{c}\,x)$ and $B(\mathsf{c}\,x) = B\,n$. Else if x is fresh in n then $\mathsf{gel}\,n\,x$ typechecks and the reduction result is $\mathsf{ind}_{\mathsf{Nm}}\,x\,n\,b_0\,b_1 = b_1\,(\mathsf{gel}\,n\,x) : B(\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x))$ where forg is the function defined in Section 3.2.1. And we have:

$$
\begin{aligned}
&\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x)\\
&= \mathsf{ext}\,[\lambda(g' : (x : @\mathsf{N}) \multimap \mathsf{Gel}\,\mathsf{Nm}\,x).\lambda(\_ : @\mathsf{N}).\mathsf{ung}\,g']\,x\,(\mathsf{gel}\,n\,x) && (\text{def.})\\
&= (\lambda(\_ : @\mathsf{N}).\,\mathsf{ung}\,(\lambda y.\,\mathsf{gel}\,n\,y))\,x && (\textsc{ext}\beta)\\
&= \mathsf{ung}\,(\lambda y.\,\mathsf{gel}\,n\,y) = n && (\multimap\beta, \textsc{Gel}\beta)
\end{aligned}
$$

The $\textsc{ext}\beta$ rule triggers because (1) by assumption, x is fresh in n, i.e. n does not mention x, nor cartesian variables strictly to the right of x (2) thus the term $\mathsf{gel}\,n\,x$ mentions x but no cartesian variables declared later. The latter condition is exactly the condition under which this ext call can reduce. To that end, x is captured in the term $\mathsf{gel}\,n\,x$ leading to a term $g' := \lambda y.\,\mathsf{gel}\,n\,y$ appearing on the second line.

SAP at Nm We now prove that $1 + \mathsf{Nm} \simeq @\mathsf{N} \multimap \mathsf{Nm}$. We take inspiration from [CH21] who developed a relational encode-decode technique to prove the SRP at data types. The type Nm is defined via an induction principle, and a similar technique can be applied.

```
decode : 1 + Nm → @N ⊸ Nm
decode (inl tt) = λ x. c x   ;   decode (inr n) = λ _. n

t1 : (@N ⊸ Nm) → (x : @N) ⊸ 1 + Gel Nm x
t1 n' x = indNm x (n' x) (inl tt) (λ (g : Gel Nm x). inr g)

t2pre : (x : @N) ⊸ (1 + Gel Nm x) → Gel (1 + Nm) x
t2pre x (inl tt) = gel (inl tt) x
t2pre x (inr g)  = ext [λ g'. λ y. gel (inr (ung g')) y] x g
```

```
t2 : ((x : @N) ⊸ 1 + Gel Nm x) → (x : @N) ⊸ Gel (1 + Nm) x
t2 = λ s' x. t2pre x (s' x)

encode : (@N ⊸ Nm) → 1 + Nm
encode n' = ung (t2 (t1 n'))
```

It remains to prove the roundtrip equalities. The roundtrip for $s : 1 + \mathsf{Nm}$ is obtained by induction on s. If $s = \mathsf{inl}\,\mathsf{tt}$ then $(\mathsf{ung} \circ t_2 \circ t_1 \circ \mathsf{decode})\, s = (\mathsf{ung} \circ t_2 \circ t_1)(\lambda x.\, \mathsf{c}\, x) \overset{\mathsf{Nm}\beta_0}{=} (\mathsf{ung} \circ t_2)(\lambda x.\, \mathsf{inl}\,\mathsf{tt}) = \mathsf{ung}\,(\lambda x.\, \mathsf{gel}\,(\mathsf{inl}\,\mathsf{tt})\, x) = \mathsf{inl}\,\mathsf{tt} = s$. If $s = \mathsf{inr}\, n$ then $(\mathsf{ung} \circ t_2 \circ t_1 \circ \mathsf{decode})\, s = (\mathsf{ung} \circ t_2 \circ t_1)(\lambda x.\, n) \overset{\mathsf{Nm}\beta_1}{=} (\mathsf{ung} \circ t_2)(\lambda x.\, \mathsf{inr}(\mathsf{gel}\, n\, x)) = \mathsf{ung}\,(\lambda x.\, \mathsf{ext}\,[\lambda g' y.\, \mathsf{gel}\,(\mathsf{inr}(\mathsf{ung}\, g'))\, y]\, x\,(\mathsf{gel}\, n\, x)) \overset{\textsc{ext}\beta}{=} \mathsf{ung}\,(\lambda x.\, \mathsf{gel}\,(\mathsf{inr}\, n) x) = \mathsf{inr}\, n = s$.

The other roundtrip is the last type in the following chain of equivalences. The first type is a sufficient condition that can be understood as a propositional η-rule for Nm.

$$\begin{aligned}
(x : @\mathsf{N}) \multimap (n : \mathsf{Nm}) \to (n \equiv_{\mathsf{Nm}} \mathsf{ext}\,[\mathsf{decode} \circ \mathsf{encode}]\, x\, n) &\qquad \simeq_{(\mathrm{SAP}_{\to})} \\
(n' : @\mathsf{N} \multimap \mathsf{Nm}) \to (x : @\mathsf{N}) \multimap (n' x \equiv_{\mathsf{Nm}} (\mathsf{decode} \circ \mathsf{encode})\, n'\, x) &\qquad \simeq_{(\mathrm{SAP}_{\equiv})} \\
(n' : @\mathsf{N} \multimap \mathsf{Nm}) \to n' \equiv (\mathsf{decode} \circ \mathsf{encode})\, n' &
\end{aligned}$$

The ext in the first line vanishes in the second because $\textsc{ext}\beta$ triggers. We prove the sufficient condition. Let $(x : @\mathsf{N}), (n : \mathsf{Nm})$ in context. We reason by name induction. If $n = \mathsf{c}\, x$ then the right-hand side is $\mathsf{ext}\,[\mathsf{decode} \circ \mathsf{encode}]\, x\,(\mathsf{c}\, x) \overset{\textsc{ext}\beta}{=} ((\mathsf{decode} \circ \mathsf{encode})\, \mathsf{c})\, x$. From the proof above we know that $\mathsf{encode}\, \mathsf{c} = \mathsf{inl}\,\mathsf{tt}$, and by definition $\mathsf{decode}\,(\mathsf{inl}\,\mathsf{tt}) = \mathsf{c}$. So both sides of the equality are equal to $\mathsf{c}\, x$. Else, formally we must provide a b_1 argument to $\mathsf{ind}_{\mathsf{Nm}}$. We define $b_1 := b_1'\, x$ where the type of b_1' is the first type in this SAP derivation, performed in the empty context and where $[\ldots] := \mathsf{decode} \circ \mathsf{encode}$. For space reasons, $(x : @\mathsf{N}) \multimap$ is rather written $x \multimap$ in the derivation.

$$\begin{aligned}
& x \multimap (g : \mathsf{Gel}\,\mathsf{Nm}\, x) \to \mathsf{forg}\, x\, g \equiv \mathsf{ext}\,[\ldots]\, x\,(\mathsf{forg}\, x\, g) \\
\simeq\ & (g' : x \multimap \mathsf{Gel}\,\mathsf{Nm}\, x) \to x \multimap \mathsf{forg}\, x\,(g'\, x) \equiv \mathsf{ext}\,[\ldots]\, x\,(\mathsf{forg}\, x\,(g'\, x)) \\
\simeq\ & (n : \mathsf{Nm}) \to x \multimap \mathsf{forg}\, x\,(\mathsf{gel}\, n\, x) \equiv \mathsf{ext}\,[\ldots]\, x\,(\mathsf{forg}\, x\,(\mathsf{gel}\, n\, x))
\end{aligned}$$

By a computation above, $\mathsf{forg}\, x\,(\mathsf{gel}\, n\, x) = n$ since x is fresh in n. The right-hand side computes to n as well by $\textsc{ext}\beta$ and definition of $[\ldots] = \mathsf{decode} \circ \mathsf{encode}$.

Nominal data types When looking at a specific example of a nominal data type D we temporarily extend the type system with the rules of D. This includes the dependent eliminator of D. For example in the case of the Ltm type of Section 3.1 we add a rule that entails the existence of this closed program:

Parameters	K'	$(x : @\mathsf{N}) \multimap K$
$A : @\mathsf{N} \multimap \mathcal{U}$, $B : (x : @\mathsf{N}) \multimap A\,x \to \mathcal{U}$.	$(a' : (x : @\mathsf{N}) \multimap A\,x) \to (x : @\mathsf{N}) \multimap B\,x\,(a'\,x)$	$(x : @\mathsf{N}) \multimap (a : A\,x) \to B\,x\,a$
	$(a' : (x : @\mathsf{N}) \multimap A\,x) \times ((x : @\mathsf{N}) \multimap B\,x\,(a'\,x))$	$(x : @\mathsf{N}) \multimap (a : A\,x) \times (B\,x\,a)$
	$\mathcal{U}$	$(x : @\mathsf{N}) \multimap \mathcal{U}$
$A : \mathcal{U}, a_0, a_1 : @\mathsf{N} \multimap A$.	$a_0 \equiv_{@\mathsf{N} \multimap A} a_1$	$(x : @\mathsf{N}) \multimap a_0\,x \equiv a_1\,x$
	$1 + \mathsf{Nm}$	$(x : @\mathsf{N}) \multimap \mathsf{Nm}$
$A : \mathcal{U}$	A	$(x : @\mathsf{N}) \multimap \mathsf{Gel}\,A\,x$
$A : \mathcal{U}, y \neq x$	$\mathsf{Gel}\,((x : @\mathsf{N}) \multimap A)\,y$	$(x : @\mathsf{N}) \multimap \mathsf{Gel}\,A\,y$
$A : (x : @\mathsf{N}) \multimap (y : @\mathsf{N}) \multimap \mathcal{U}$	$(y : @\mathsf{N}) \multimap (x : @\mathsf{N}) \multimap A\,x\,y$	$(x : @\mathsf{N}) \multimap (y : @\mathsf{N}) \multimap A\,x\,y$

Table 3.2: The Structure Abstraction Principle

```
indLtm : (P : Ltm → 𝒰) → ((n : Nm) → P (var n)) →
  (∀ a b. P a → P b → P (app a b)) →
  (∀ g. ((x : @N) ⊸ P (g x)) → P (lam g)) → ∀ t. P t
```

3.2.3 The Structure Abstraction Principle (SAP)

For types and terms As hinted above, PNTT validates the Structure Abstraction Principle (SAP). Similar to the SIP of HoTT/UF, or the SRP of binary PTT [VND24], the SAP defines how each type former commutes with the (nullary) Bridge type former. Table 3.2 lists several SAP instances, including the ones we have encountered so far. Each instance is[2] of the form $\forall \cdots . K' \simeq ((x : @\mathsf{N}) \multimap K)$, where K, K' may depend on some parameters. Note that the instances in Table 3.2 involve primitive type formers K. In order to obtain the SAP instance of a composite type K, we can combine the SAP instances of the primitives used to define K (this was done e.g. in Section 3.2.2 when proving the propositional η-rule of Nm, and is further discussed below). Types of DTT can contain terms and accordingly there exists a SAP for terms. Let $\mathsf{SAP}_K : K' \simeq (@\mathsf{N} \multimap K)$ be a SAP instance. The SAP of the term $k : K$ is a pair (k', p) where $k' : K'$ and $p : \mathsf{SAP}_K\,k' \equiv \lambda(\_ : @\mathsf{N}).k$, i.e. k' is the reflexivity bridge $\mathsf{rfl}\,k := \lambda(\_ : @\mathsf{N}).k$ of $k : K$ through the SAP_K equivalence, up to propositional equality.

Relationship to the translation For types, it is basically the case that the type K' provided by the SAP at K is $[K]_0 : \mathcal{U}$, the recursively and externally defined nullary parametricity translation of K. The n-ary translation is briefly discussed in Section 3.1. Note that in practice it is sometimes useful to consider SAP instances where K' is a type different but equivalent to $[K]_0$, but this detail can be ignored

[2]Orienting SAP_K this way enables $\mathsf{SAP}_{\mathcal{U}} = \mathsf{Gel}$ and $\mathsf{SAP}_{\Pi} = \mathsf{ext}$.

for clarity. For terms, it is also the case that the SAP at $k : K$ provides a k' which is $[k]_0 : [K]_0$, the recursively defined nullary translation of $k : K$ (analogously it is sometimes useful to let k' be a term different but propositionally equal to $[k]_0$). From that perspective, $\mathsf{SAP}_K : [K]_0 \simeq @\mathsf{N} \multimap K$ says that the Bridge type of K is its translation up to an equivalence, and the SAP of a term $k : K$ says that $[k]_0 \equiv \mathsf{SAP}_K^{-1}\ (\lambda(\_ : @\mathsf{N}).k)$, i.e. the reflexivity bridge at $k : K$ is its translation up to a propositional equality. It is useful to give a name to the latter right-hand side. If a SAP instance is fixed for K we define the *observational parametricity* of a term $k : K$ to be $[k]_0^{\mathcal{O}} := \mathsf{SAP}_K^{-1}\ (\lambda(\_ : @\mathsf{N}).k)$. The various parametricity mappings for types and terms can be informally summarized as follows, where $\overset{m}{\to}$ denotes a metatheoretical function.

$$
\begin{array}{rcl}
[-]_0 : \mathcal{U} \overset{m}{\to} \mathcal{U} & ; & [-]_0, [-]_0^{\mathcal{O}} : \{K : \mathcal{U}\} \overset{m}{\to} K \overset{m}{\to} [K]_0 \\
@\mathsf{N} \multimap - : \mathcal{U} \to \mathcal{U} & ; & \mathsf{rfl} : \{K : \mathcal{U}\} \to (k : K) \to @\mathsf{N} \multimap K \\
\mathsf{SAP} : (K : \mathcal{U}) \overset{m}{\to} [K]_0 \simeq @\mathsf{N} \multimap K & ; & \mathsf{SAP} : \{K : \mathcal{U}\} \overset{m}{\to} (k : K) \overset{m}{\to} [k]_0 \equiv [k]_0^{\mathcal{O}}
\end{array}
$$

As an example, we compute the observational parametricity of a function $f : A \to B$. Suppose A and B are types with SAP instances $A' \simeq (@\mathsf{N} \multimap A)$ and $B' \simeq (@\mathsf{N} \multimap B)$. The SAP at $A \to B$ is the composition $\mathsf{SAP}_{A\to B}^{-1} : (@\mathsf{N} \multimap (A \to B)) \simeq ((@\mathsf{N} \multimap A) \to @\mathsf{N} \multimap B) \simeq A' \to B'$. The map $\mathsf{SAP}_{A\to B}^{-1}$ is given by $\lambda f'.\, \mathsf{SAP}_B^{-1} \circ (\mathsf{ext}^{-1}\ f') \circ \mathsf{SAP}_A$ where $\mathsf{ext}^{-1} f' = \lambda(a' : @\mathsf{N} \multimap A).\, \lambda(x : @\mathsf{N}).\, f'\, x\, (a'\, x)$. With this choice of SAP instances, we have $[f]_0^{\mathcal{O}} : A' \to B'$ and $[f]_0^{\mathcal{O}} = \mathsf{SAP}_B^{-1} \circ (@\mathsf{N} \multimap f) \circ \mathsf{SAP}_A$ where $(@\mathsf{N} \multimap f) = \lambda a'\, x.\, f\, (a' x)$ is the action of f on bridges. So the SAP for a function $f : A \to B$ asserts that its action on bridges agrees with its translation, up to the SAP at A, B.

The SAP of a composite type K is obtained by *manually* applying the SAP rules from Table 3.2. This process will produce a translation K' that is equivalent to $@\mathsf{N} \multimap K$. In the binary case, we have shown in earlier work [VND24] that this process can be systematized, by organizing types and terms together with their SRP instances into a (shallowly embedded) type theory called ROTT. For types K and terms $k : K$ that fall in the ROTT syntax, the parametricity translations and SAP instances can be derived straightforwardly. Although we believe the same process applies here, we have not constructed the corresponding DSL, instead applying SAP instances manually in examples.

3.2.4 About Semantics

Similar to our theory PNTT, our model is a simple extension of Cavallo and Harper's presheaf model [CH21] with the arity of parametricity changed from 2 to 0. This model is $\mathsf{Psh}(\boxslash_2 \times \square_0)$, where $\boxslash_2$ is the binary cartesian cube category [Ang+21a] for path dimensions, and $\square_0$ is the 0-ary affine cube category for bridge dimensions. Concretely $\square_0$ is the opposite of the category of finite ordinals and injections. We

keep this section overall succinct, not for lack of formal understanding, because these semantics are a well-understood combination of established models of type theory [Hof97; HS97; Ang+21a; BCM15; CH21].

In this model the type Nm is interpreted as the Yoneda embedding of the base object with 1 bridge dimension and no path dimensions. The elimination and computation rules for this type are based on a semantic isomorphism $x : @\mathsf{N} \vdash \mathsf{Nm} \cong 1 + \mathsf{Gel}\,\mathsf{Nm}\,x$ which is straightforwardly checked. In fact, the above (taken as an equivalence) is equivalent (by several invocations of the SAP) to $(@\mathsf{N} \multimap \mathsf{Nm}) \simeq 1 + \mathsf{Nm}$, an equivalence semantically proven by Hofmann [Hof99]. Kan fibrancy of Nm is semantically trivial, since we can tell from the base category that any path in Nm will be constant. As for computation of Kan operations at Nm, we propose to wait until all arguments reduce to $\mathsf{c}\,x$ for the same affine name x, in which case we return $\mathsf{c}\,x$. This is in line with how Kan operations for positive types with multiple constructors are usually reduced [Coh+17; Ang+21a]. Regarding nominal data types, which we only consider through examples in this paper, we take the viewpoint that these arise from an interplay between the usual type formers, bridge types, Nm, and an initial algebra operation for 'nominal strictly positive functors'. We do not attempt to give a categorical description of such functors or prove that they have initial algebras. The Kan operation will reduce recursively and according to the Kan operations of all other type formers involved.

We note that $\square_0$ is the base category (carrier of the site) of the Schanuel sheaf topos [Pit13, §6.3] which is equivalent to the category of nominal sets [Pit13], used to model FreshMLTT [PMD14]. The sheaf condition requires that presheaves $\Gamma : \square_0^{\mathsf{op}} \to \mathrm{Set}$ preserve pullbacks, i.e. compatible triples in $\Gamma_{U \uplus V} \to \Gamma_{U \uplus V \uplus W} \leftarrow \Gamma_{U \uplus W}$ have a unique preimage in Γ_U. Conceptually, if a cell $\gamma \in \Gamma_{U \uplus V \uplus W}$ is both fresh for V and for W, then it is fresh for $V \uplus W$, with unique evidence. This property is not available internally in nullary PTT, nor do we have the impression that it is important to add it, so we content ourselves by modeling types as *pre*sheaves over $\square_0$.

3.3 Nominal Primitives for Free

In this section, we argue that the central features in a number of earlier nominal dependent type systems can essentially be recovered in nullary PTT. One notable exception is nominal pattern matching (discussed in the next section), which is recovered in our extension PNTT. Concretely, we consider here Shinwell, Pitts and Gabbay's FreshML [SPG03], Schöpp and Stark's bunched nominal type theory BNTT [SS04; Sch06], Cheney's $\lambda^{\Pi\mathcal{N}}$ [Che12] and Pitts, Matthiesen and Derikx's FreshMLTT [PMD14]. The central features we identify there, are: existential and universal name quantification, a type former expressing freshness for a given name, name swapping, and locally scoped names. Existential and universal name quantification are known to be equivalent in the usual (pre)sheaf or nominal set semantics of nominal type theory, but generally have quite different typing rules: the

former is an existential type former with pair-like constructor and matching eliminator (opening the door to matching more deeply), whereas the latter is a universal type operated through name abstraction and application (getting in the way of matching more deeply). We note that some systems (FreshMLTT, $\lambda^{\Pi\mathcal{U}}$) support multiple name types, something we could also easily accommodate, but leave out so as not to distract from the main contributions. BNTT even allows substructural quantification and typal freshness for arbitrary closed types (rather than just names), which is not something we intend to support and which inherently seems to require a bunched context structure. In what follows, we will speak of 'our rules', not to claim ownership (as they are inherited from Bernardy, Coquand and Moulin [BCM15] and Cavallo and Harper [CH21]), but to distinguish them from the other systems.

Universal name quantification Universal name quantification is available in BNTT, $\lambda^{\Pi\mathcal{U}}$ and FreshMLTT. In our system, it is done using the nullary bridge type $(x : @\mathsf{N}) \multimap A$, whose rules are given in Fig. 3.1. The rules $\multimap$F and $\multimap$I correspond almost perfectly with the other three systems; for BNTT we need to keep in mind that our context extension with a fresh name, is semantically a monoidal product. The application rules in $\lambda^{\Pi\mathcal{U}}$ and BNTT also correspond almost precisely to $\multimap$E, but the one in BNTT is less algorithmic than ours. Specifically, the BNTT bunched application rule takes a function $\Theta \vdash f : (x : T) \multimap A\,x$ and an argument $\Delta \vdash t : T$ and produces $f\,t : A\,t$ in a non-general context $\Theta * \Delta$. Our rule $\multimap$E (inherited from [BCM15; CH21]) improves upon this by taking in an arbitrary context Γ and *computing* a context $\Gamma\backslash x$ such that there is a morphism $\Gamma \to (\Gamma\backslash x, (x : @\mathsf{N}))$. The application rule in FreshMLTT is similar, but uses definitional freshness (based on a variable swapping test) to ensure that the argument is fresh for the function.

Typal freshness A type former expressing freshness is available in BNTT (called the free-from type). FreshMLTT uses definitional freshness instead. In nullary PTT, elements of A for which x is fresh are classified by the type $\mathsf{Gel}\,A\,x$. BNTT's free-from types only apply to closed types A, so GelF is more general. BNTT's introduction rule corresponds to GelI, which is however again more algorithmic. BNTT's elimination rule is explained in terms of single-hole bunched contexts [Sch06, §4.1.1] which specialize in our setting (where the only monoidal product is context extension with a name) to contexts with a hole up front. Essentially then, the BNTT rule can be phrased in nullary PTT as

$$\frac{\Gamma, x : @\mathsf{N}, z : \mathsf{Gel}\,B\,x, \Theta \vdash T\ \mathsf{type} \qquad \Gamma, y : B, x : @\mathsf{N}, \Theta[\mathsf{gel}\,y\,x/z] \vdash t : T[\mathsf{gel}\,y\,x/z]}{\Gamma, x : @\mathsf{N}, z : \mathsf{Gel}\,B\,x, \Theta \vdash t' : T}$$

where Γ must be empty, together with β- and η-rules establishing that the above operation is inverse to applying the substitution $[\mathsf{gel}\, y\, x/z]$.[3] We can in fact accommodate the rule for non-empty Γ. Without loss of generality, we can assume Θ is empty: by abstraction/application, we can subsume Θ in T.[4] We then have an equivalence

$$\begin{aligned}
&(y : B) \to ((x : @\mathsf{N}) \multimap T[\mathsf{gel}\, y\, x/z]) \\
\simeq\ &(y : (x : @\mathsf{N}) \multimap \mathsf{Gel}\, B\, x) \to ((x : @\mathsf{N}) \multimap T[y\, x/z]) \\
\simeq\ &(x : @\mathsf{N}) \multimap (z : \mathsf{Gel}\, B\, x) \to T
\end{aligned}$$

where in the first step, we precompose with the gel/ung isomorphism (the SAP for Gel), and in the second step, we apply the ext equivalence (the SAP for functions).

Existential name quantification Existential name quantification is available in FreshML and BNTT. We first discuss how we can accommodate the BNTT rules, and then get back to FreshML. The BNTT existential quantifier is just translated to the nullary bridge type $(x : @\mathsf{N}) \multimap A$ again, i.e. both quantifiers become definitionally the same type in nullary PTT. BNTT's introduction rule follows by applying a function $\mathsf{bind} : (x : @\mathsf{N}) \multimap B\, x \to \mathsf{Gel}\, ((w : @\mathsf{N}) \multimap B\, w)\, x$ which is obtained from the identity function on $(w : @\mathsf{N}) \multimap B\, w$ by the SAP for functions and Gel. BNTT's elimination rule essentially provides a function $\mathsf{matchbind} : ((x : @\mathsf{N}) \multimap B\, x \to \mathsf{Gel}\, C\, x) \to (((x : @\mathsf{N}) \multimap B\, x) \to C)$ such that if we apply $\mathsf{matchbind}\, f$ under Gel to $\mathsf{bind}\, x\, b$, then we obtain $f\, x\, b$. Again, the SAP for Gel and functions reveals that the source and target of matchbind are equivalent.

The non-dependently typed system FreshML has a similar type former, but lacks any typal or definitional notion of freshness. Their introduction rule then follows by postcomposing the above bind with the function forg that forgets freshness (Section 3.2.1). If we translate FreshML's declarations $\Gamma \vdash d : \Delta$ to operations that convert terms $\Gamma, \Delta \vdash t : T$ to terms $\Gamma \vdash t\{d\}$ (not necessarily by substitution), then we can translate their declaration $\Gamma \vdash \mathtt{val}\, \langle x \rangle y = e : (x : @\mathsf{N}, y : B)$, where e is an existential pair, as $\mathsf{matchdecl}\, e$ where $\mathsf{matchdecl} : (e : @\mathsf{N} \multimap B) \to (t : @\mathsf{N} \multimap B \to T) \to (@\mathsf{N} \multimap T)$. This function is obtained by observing that the second argument type, by the SAP for functions, is equivalent to $(@\mathsf{N} \multimap B) \to (@\mathsf{N} \multimap T)$. It may be surprising that the result has type $@\mathsf{N} \multimap T$ rather than just T; this reflects the fact that FreshML cannot enforce freshness, and is justified by the fact that it allows arbitrary declaration of fresh names via the declaration $\Gamma \vdash \mathtt{fresh}\, x : (x : @\mathsf{N})$, which we do not support.

Swapping names The name swapping operation is available in FreshML and FreshMLTT. We can accommodate the full rule of FreshML (which is not dependently

[3] The original rule immediately subsumes a substitution $\Delta \to (x : @\mathsf{N}, z : \mathsf{Gel}\, A\, x)$.

[4] This may raise questions about preservation of substitution w.r.t. Θ. However, it is not at all standard for dependently typed elimination rules to guarantee preservation of substitution w.r.t. a telescope depending on the eliminee.

typed) and a restricted version of the rule in FreshMLTT, where we allow the type of the affected term to depend on the names being swapped, but other than that, only on variables fresh for those names. We simply use the function $\mathsf{swap} : (x\,y : @\mathsf{N}) \multimap T\,x\,y \to T\,y\,x$ whose type is equivalent to $((x\,y : @\mathsf{N}) \multimap T\,x\,y) \to ((x\,y : @\mathsf{N}) \multimap T\,y\,x)$ by the SAP, and the latter is clearly inhabited by $\lambda\, t\, x\, y.\, t\, y\, x$. Note that name swapping needs to happen w.r.t. affine names: swapping the cartesian names $x, y : \mathsf{Nm}$ in the term $\mathsf{app}\,(\mathsf{app}\,(\mathsf{var}\,x)\,(\mathsf{var}\,y))\,(\mathsf{var}\,z) : \mathsf{Ltm}$ does not commute with contraction $-[x/x, x/z]$. Finally, we note that the aforementioned restriction on the type can be mitigated in an ad hoc manner, because types $T : (x\,y : @\mathsf{N}) \multimap \Delta \to \mathcal{U}$ (where Δ denotes any telescope consisting of both affine names and non-affine variables) are by the SAP in correspondence with types $\Delta'' \to (x\,y : @\mathsf{N}) \multimap \mathcal{U}$ for a different telescope Δ''. This however does not imply that we can accommodate the full FreshMLTT name swapping rule in a manner that commutes with substitution. We expect that this situation can be improved by integrating ideas related to the transpension type [ND24; Nuy25] into the current system. Lastly, using swap, we can follow Pitts et al. [PMD14] in defining non-binding abstraction $\langle x\rangle - = \lambda\, a\, y.\, \mathsf{swap}\, x\, y\, a : A\, x \to (y : @\mathsf{N}) \multimap A\, y$, where A can depend only on variables fresh for x.

Locally scoped names Locally scoped names [Ode94; Pit10] are available in FreshMLTT, by the following rule on the left, and allow us to spawn a name from nowhere, provided that we use it to form a term that is fresh for it:

$$\frac{\Gamma, x : @\mathsf{N} \vdash T\ \mathsf{type} \quad \Gamma, x : @\mathsf{N} \vdash t : T \quad x \text{ is fresh for } t : T}{\Gamma \vdash \nu\, x.\, t : \nu\, x.\, T} \qquad \Gamma, x : @\mathsf{N} \vdash \nu\, x.\, t = t : T \quad \text{Added: } \nu\, x.\, t = t \text{ if } x \text{ is not free in } t. \qquad \frac{\Gamma \vdash S\ \mathsf{type} \quad \Gamma, x : @\mathsf{N} \vdash t : S \quad x \text{ is fresh for } t : S}{\Gamma \vdash \nu\, x.\, t : S}$$

It is used [Pit14] in FreshMLTT to define e.g.

$$\lambda\, c'.\, \nu\, x.\, \mathsf{case}\,(c'\, x)\, \begin{cases} \mathsf{inl}\, a \mapsto \mathsf{inl}\, \langle x\rangle a \\ \mathsf{inr}\, b \mapsto \mathsf{inr}\, \langle x\rangle b \end{cases}$$

Of type $(@\mathsf{N} \multimap A + B) \to (@\mathsf{N} \multimap A) + (@\mathsf{N} \multimap B)$. It has a computation rule which says that as soon as x comes into scope again, we can drop the ν-binder. Combined with α-renaming, this means $\nu\, x.$ says 'let x be any name we have in scope, or a fresh one, it doesn't matter'. In particular, $\nu\, x.-$ is idempotent. Before translating to nullary PTT, we add another equation rule, which says that we can drop $\nu\, x.-$ if x is not used freely at all (in the example above, x is used freely but freshly). This way, the FreshMLTT ν-rule above becomes equivalent to the one to its right. Indeed, the functions $T \mapsto \nu\, x.\, T$ and $S \mapsto S$ now constitute an isomorphism between types that are and are not dependent on $x : @\mathsf{N}$. The advantage of the rule on the right is that it is not self-dependent. In nullary PTT, we express freshness using Gel, suggesting the following adapted rules:

$$\frac{\Gamma, x : @\mathsf{N} \vdash t : \mathsf{Gel}\, S\, x}{\Gamma \vdash \nu\, x.\, t : S} \qquad \begin{array}{l}\Gamma, x : @\mathsf{N} \vdash \mathsf{gel}\, (\nu\, x.\, t)\, x = t : \mathsf{Gel}\, S\, x \\ \nu\, x.\, \mathsf{gel}\, t\, x = t \text{ (where } x \text{ cannot be free in } t \text{ by GelI).}\end{array}$$

In this formulation, it is now clear that $\nu x.\, t$ can be implemented as $\mathsf{ung}\ (\lambda x.\, t)$, while the two computation rules follow from Gelη and Gelβ.

3.4 Nominal Pattern Matching

In this section we provide concrete examples of functions $D \to E$ defined by recursion on a nominal data type D, within PNTT, explained in Section 3.2.

3.4.1 Patterns That Bind

Some nominal frameworks with existential name-abstraction types [SPG03] provide a convenient user interface to define functions $f : D \to E$ out of a nominal data type. The user can define f by matching on patterns that bind names (e.g. eqabs in Section 3.1). We explain how this feature is indirectly recovered in PNTT.

The following nominal data type [BP09] is the nominal syntax of the π-calculus [MPW92], a formal language whose expressions represent concurrently communicating processes. The constructors stand for: terminate, silent computation step, parallelism, non-determinism, channel allocation, receiving[5], and sending. The arguments of (output) type Proc can be thought of as continuations, e.g. nu prepends a channel allocation instruction.

```
data Proc : 𝒰 where
  nil   : Proc
  τpre  : Proc → Proc
  par   : Proc → Proc → Proc
  sum   : Proc → Proc → Proc
  nu    : (@N ⊸ Proc) → Proc
  inp   : Nm → (@N ⊸ Proc) → Proc
  out   : Nm → Nm → Proc → Proc
```

An example of a process is $\mathsf{par}\, (\mathsf{out}\, a\, b\, q)\, (\mathsf{inp}\, a\, (\lambda(x : @\mathsf{N}).\, p' x))$. The first argument of par emits a name b on channel a and continues with q. Simultaneously, the second argument waits for a name x on channel a and continues with $p'\, x$. The expectation is that such an expression should reduce to $\mathsf{par}\, q\, (p'\{b/x\})$ where $p'\{b/x\}$ replaces occurrences of $(x : @\mathsf{N})$ in the body of p' by the name $b : \mathsf{Nm}$. Note that the application $p'\, b$ does not typecheck and that this substitution operation called nsub below must be defined recursively, as done in [BP09]. The following is an informal definition of nsub where some patterns bind (bridge) variables.

[5] Binders have arguments $@\mathsf{N} \multimap -$ and not $\mathsf{Nm} \to -$ since the latter could lead to exotic process terms checking the bound name for equality to other names.

```
nsub : Nm → (@N ⊸ Proc) → Proc
nsub b (λ x. par (u' x) (v' x)) = par (nsub b u') (nsub b v')
...  -- nil, τpre, sum similar
nsub b (λ x. nu (q' x))         = nu  (λ y. nsub b (λ x. q' x y))
nsub b (λ x. inp a (q' x))      = inp a (λ y. nsub b (λ x. q' x y))  -- (0)
nsub b (λ x. inp (c x) (q' x))  = inp b (λ y. nsub b (λ x. q' x y))  -- (1)
...
```

Note how patterns (0) and (1) match on a binding pattern of the form $\lambda x.\,\mathsf{inp}\ m\ (q'\ x)$, and cover the case where m is different, resp. equal to the variable being substituted. The informal nsub function reduces accordingly.

More formally in our system we define nsub by using the SAP at Proc, so $\mathsf{nsub}\ b$ is the following composition $(@\mathsf{N} \multimap \mathsf{Proc}) \xrightarrow{\simeq} \mathsf{AProc} \longrightarrow \mathsf{Proc}$. The SAP at Proc asserts that $(@\mathsf{N} \multimap \mathsf{Proc})$ is equivalent to AProc, the nullary translation of Proc (its constructors use a suggestive $[-]_0$ notation):

```
data AProc : 𝒰 where
  [nil]₀    : AProc
  [τpre]₀   : AProc → AProc
  [par]₀    : AProc → AProc → AProc
  [sum]₀    : AProc → AProc → AProc
  [nu]₀     : (@N ⊸ AProc) → AProc
  [inp]₀    : 1 + Nm → (@N ⊸ AProc) → AProc
  [out]₀    : 1 + Nm → 1 + Nm → AProc → AProc
```

We can then define $\mathsf{nsub}' : \mathsf{Nm} \to \mathsf{AProc} \to \mathsf{Proc}$ by induction on the second argument. Indeed the informal clauses (0), (1) can be translated into formal ones using the $[\mathsf{inp}]_0$ constructor. More precisely, clause (1) uses the function $[\mathsf{inp}]_0\ (\mathsf{inl}\ \mathsf{tt})$ of type $(@\mathsf{N} \multimap \mathsf{AProc}) \to \mathsf{AProc}$ and clause (0) uses the function $\lambda(n : \mathsf{Nm}).\ [\mathsf{inp}]_0\ (\mathsf{inr}\ n)$ of type $\mathsf{Nm} \to (@\mathsf{N} \multimap \mathsf{AProc}) \to \mathsf{AProc}$.

3.4.2 A HOAS Example

In the example sketched below, we connect the nominal syntax of the untyped lambda calculus (ULC) to a nominal, higher-order abstract syntax (HOAS) representation. The example is performed within PNTT plus a single (standard) binary parametricity axiom. Our proof is a port to PNTT of an existing argument by Atkey [Atk09], which uses external *Kripke* binary parametricity to connect the De Bruijn (non-nominal) syntax of ULC to a (non-nominal) HOAS representation.

This example serves two purposes. The first is to illustrate, by porting a complex argument from one setting to the other, how the combination of nullary (for nominal reasoning) and binary parametricity allows for a reasoning style similar to that of Kripke binary parametricity, and moreover with closely related denotational semantics. This topic is elaborated further at the end of this section.

Secondly, the example illustrates that more often than not, proving the correctness of a function $f : D \to E$ defined by recursion out of a nominal data type D requires computing the parametricity translation of f. Indeed, for $\mathsf{bind} : (@\mathsf{N} \multimap D) \to D$ a binder (constructor) of D, the β-rule of D says that the redex $f(\mathsf{bind}\, d')$ computes to a term involving $@\mathsf{N} \multimap f$, the action of f on bridges (see Section 3.2.3). So proving f correct often requires characterizing $@\mathsf{N} \multimap f$ up to propositional equality. This characterization is the SAP for f (also in Section 3.2.3) $[f]_0 \equiv_{[D]_0 \to [E]_0} [f]_0^{\mathcal{O}}$, where $[f]_0$ is the recursive translation of f and $[f]_0^{\mathcal{O}} := \mathsf{SAP}_E^{-1} \circ (@\mathsf{N} \multimap f) \circ \mathsf{SAP}_D$ the observational parametricity of f.

SAP proofs for composite terms like f are unwieldy because one has to show why the SAP instances for the primitives in f compose to the global one and this involves manually reproducing the structure of the term f at the level of SAP proofs. For this reason some instances of the SAP for terms are assumed to hold below, without proof (see paragraph "Roundtrip at Ltm_j"). The unwieldiness of interval-based PTTs is a good selling point for *observational* parametric type theories (see Section 3.5), where the SAP is not needed since a first-class recursive translation attempts to replace $@\mathsf{N} \multimap -$ altogether.

As a side note regarding the above function $f : D \to E$, its domain D could feature a binder of type $(@\mathsf{N} \multimap \ldots \multimap @\mathsf{N} \multimap D) \to D$ binding m fresh names. In that case proving the correctness of f may involve computing its m-fold iterated nullary parametricity, for the same reasons as above. No such examples are investigated here.

Let us now get to the actual argument. The nominal syntax of ULC is expressed as the following data type family Ltm, parametrized by a natural number j : `nat`.

```
data Ltm (j : nat) : 𝒰 where
  env : Fin j → Ltm j
  var : Nm → Ltm j
  app : Ltm j → Ltm j → Ltm j
  lam : (@N ⊸ Ltm j) → Ltm j
```

The type Fin j is the finite type with j elements $\{0, \ldots, j-1\}$. Ltm_j is a shorthand for $\mathsf{Ltm}\, j$. The types Ltm and Ltm_1 of Section 3.1 are Ltm_0 and Ltm_1 respectively.

The corresponding HOAS representation, or encoding, is $\mathsf{HEnc}_j : \mathcal{U}$ defined below. Since it uses a Π-type we say it is a Π-encoding.

```
HMod (j : nat) = (H : 𝒰) × (Fin j → H) × (Nm → H) ×
  (H → H → H) × ((H → H) → H)

-- projections
|_| : ∀ {j}. HMod j → 𝒰
|_| M = M .fst
envOf, varOf, appOf, hlamOf = ...  -- other projections of HMod

Henc (j : nat) = (M : HMod j) → |M|
```

```
NMod (j : nat) = (H : 𝒰) × (Fin j → H) × (Nm → H) ×
  (H → H → H) × ((@N ⊸ H) → H)
```

The type of "nominal models" NMod_j is also defined as it will be useful later on. The carrier function $|-|$ and other projections are defined similarly for nominal models. Additionally we define explicit constructors $\mathsf{mkHM}_j, \mathsf{mkNM}_j$ for $\mathsf{HMod}_j, \mathsf{NMod}_j$. For instance $\mathsf{mkNM}_j : (H : \mathcal{U}) \to (\mathsf{Fin}\, j \to H) \to (\mathsf{Nm} \to H) \to (H \to H \to H) \to ((@\mathsf{N} \multimap H) \to H) \to \mathsf{NMod}_j$.

We will show that we can define maps between Ltm_j and the encoding HEnc_j in both ways, and sketch a conditional proof that they compose to the identity at Ltm_j.

```
ubd     : ∀ {j}. Henc j → Ltm j
toh     : ∀ {j}. Ltm j → Henc j
rdt-Ltm : ∀ {j}. (t : Ltm j) → ubd (toh j t) ≡ t
```

Unembedding We begin by defining the "unembedding" map denoted by ubd, which has a straightforward definition that does not involve nominal recursion. The name and the idea behind the definition come from [Atk09; ALY09], where Atkey et al. were interested in comparing (non-nominal) syntax and HOAS Π-encodings using a strengthened form of binary parametricity called Kripke parametricity (see end of section for a comparison with our system/model).

```
ubd {j} = λ (h : Henc j). h (LtmHMod j) where
  LtmHMod : ∀ j. HMod j
  LtmHMod j = mkHM j (Ltm j) env var app (hlamLtm j)
  hlamLtm : ∀ j. (Ltm j → Ltm j) → Ltm j
  hlamLtm j f = lam (λ (x : @N). f (var (c x)))
```

So unembedding a HOAS term h consists of applying h at Ltm_j. This is possible thanks to the fact that Ltm_j can be equipped with a higher-order operation $\mathsf{hlamLtm}_j$.

Defining the map into HOAS The other map toh_j is defined by nominal recursion. In other words it is defined using the eliminator of Ltm_j. We write the (uncurried) non-dependent eliminator as rec_j. Its type expresses that Ltm_j is the initial nominal model.

```
rec j : (N : NMod j) → Ltm j → |N|
```

Hence in order to define toh_j we need to turn its codomain HEnc_j into a nominal model. For fields in NMod_j that are not binders this is straightforward.

```
envH : Fin j → Henc j
varH : Nm → Henc j
```

$$
\begin{aligned}
&\mathsf{appH} : \mathsf{Henc\ j} \to \mathsf{Henc\ j} \to \mathsf{Henc\ j}\\
&\mathsf{envH\ k} \;\; = \lambda\ (\mathsf{M} : \mathsf{HMod\ j}).\ \mathsf{envOf\ M\ k}\\
&\mathsf{varH\ n} \;\; = \lambda\ (\mathsf{M} : \mathsf{HMod\ j}).\ \mathsf{varOf\ M\ n}\\
&\mathsf{appH\ u\ v} = \lambda\ \mathsf{M}.\ \mathsf{appOf\ M}\ (\mathsf{u\ M})\ (\mathsf{v\ M})
\end{aligned}
$$

Additionally we need to provide a function $\mathsf{lamH} : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \to \mathsf{HEnc}_j$. This is done by using the SAP at HEnc_j, i.e. the following characterization of $(@\mathsf{N} \multimap \mathsf{HEnc}_j)$.

Theorem 3. *We have* $\mathsf{ebump}_j : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \simeq \mathsf{HEnc}_{j+1}$, $\mathsf{mbump}_j : (@\mathsf{N} \multimap \mathsf{HMod}_j) \simeq \mathsf{HMod}_{j+1}$, $\mathsf{nbump}_j : (@\mathsf{N} \multimap \mathsf{NMod}_j) \simeq \mathsf{NMod}_{j+1}$ *and* $\mathsf{lbump}_j : (@\mathsf{N} \multimap \mathsf{Ltm}_j) \simeq \mathsf{Ltm}_{j+1}$.

Intuitively these results hold because $@\mathsf{N} \multimap -$ preserves all primitive type formers except Nm, which gets translated to a sum $1 + \mathsf{Nm}$. So translating each of the aforementioned types boils down to replacing Nm with $1 + \mathsf{Nm}$, which refactors to incrementing j.

Proof. We only prove mbump_j. The proofs of ebump_j and nbump_j are similar, and lbump_j is proved using an encode-decode argument [CH21; VND24] similar to the proof of $\mathsf{SAP}_{\mathsf{Nm}}$. We sometimes omit types for Σ and $\multimap$, e.g. we write $x \multimap T$ as shorthand for $(x : @\mathsf{N}) \multimap T$.

$$
\begin{aligned}
& x \multimap \mathsf{HMod}_j\\
&\simeq (H' : x \multimap \mathcal{U}) \times (x \multimap [(\mathsf{Fin}\, j \to H'x) \times \ldots]) && \mathsf{SAP}_\Sigma\\
&\simeq (H' : x \multimap \mathcal{U}) \times (x \multimap (\mathsf{Fin}\, j \to H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_\Sigma\\
&\simeq (H' : x \multimap \mathcal{U}) \times (\mathsf{env}' : \mathsf{Fin}\, j \to (x \multimap H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_{\mathsf{Fin}\, j, \to}
\end{aligned}
$$

Since $\mathsf{Fin}\, j$ is a non-nominal data type its SAP instance asserts $\mathsf{Fin}\, j \simeq (x \multimap \mathsf{Fin}\, j)$, i.e. the only bridges in $\mathsf{Fin}\, j$ are reflexive bridges [CH21; VND24]. Moving on,

$$
\begin{aligned}
&\simeq H' \times \mathsf{env}' \times (x \multimap [(\mathsf{Nm} \to H'x) \times \ldots])\\
&\simeq H' \times \mathsf{env}' \times (x \multimap (\mathsf{Nm} \to H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_\Sigma\\
&\simeq H' \times \mathsf{env}' \times ((x \multimap \mathsf{Nm}) \to (x \multimap H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_\to\\
&\simeq H' \times \mathsf{env}' \times (\mathsf{foo} : (1 + \mathsf{Nm}) \to (x \multimap H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_{\mathsf{Nm}}\\
&\simeq H' \times \mathsf{env}'_+ \times (\mathsf{var}' : \mathsf{Nm} \to x \multimap H'x) \times (x \multimap [\ldots])
\end{aligned}
$$

where $\mathsf{env}'_+ : \mathsf{Fin}\,(j+1) \to (x \multimap H'x)$ is defined as $\mathsf{env}'_+\, 0 = \mathsf{foo}\,(\mathsf{inl\ tt})$ and $\mathsf{env}'_+\,(k+1) = \mathsf{env}'k$. The next two types in $x \multimap [\ldots]$ are computed using the SAP at $\to$. Thus so far we have shown that $@\mathsf{N} \multimap \mathsf{HMod}_j$ is equivalent to $(H' : x \multimap \mathcal{U}) \times \mathsf{env}'_+ \times \mathsf{var}' \times (\mathsf{app}' : (x \multimap H'x) \to (x \multimap H'x) \to (x \multimap H'x)) \times (\mathsf{hlam}' : ((x \multimap H'x) \to (x \multimap H'x)) \to (x \multimap H'x))$. Now since the SAP at $\mathcal{U}$ is $\mathsf{Gel} : \mathcal{U} \xrightarrow{\simeq} (@\mathsf{N} \multimap \mathcal{U})$, we can do a change of variable in the above translation of HMod_j, i.e. use a variable $(K : \mathcal{U})$ instead of $(H' : x \multimap \mathcal{U})$ at the cost of replacing occurrences of H' by $\mathsf{Gel}\, K$. Pleasantly, occurrences of $(x \multimap H'x)$

become $(x \multimap \mathsf{Gel}\, K\, x)$, which is equivalent to K by the SAP for Gel types. Hence $@\mathsf{N} \multimap \mathsf{HMod}_j \simeq \mathsf{HMod}_{j+1}$. □

Next, we can define the desired operation $\mathsf{lamH} : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \to \mathsf{HEnc}_j$ as $\mathsf{lamH}\, h' = \lambda(M : \mathsf{HMod}_j).\, \mathsf{lamOf}\, M\, (\lambda(m : |M|).\, (\mathsf{ebump}_j\, h')\, (m :: M))$ where $(m :: M)$ is the HMod_{j+1} obtained out of $M : \mathsf{HMod}_j$ by pushing m onto the list $\mathsf{envOf} M$. In other words, $\mathsf{envOf}\, (m :: M)\, 0 = m$ and $\mathsf{envOf}\, (m :: M)\, (k+1) = \mathsf{envOf}\, M\, k$. All in all we proved that HEnc_j is a nominal model. Since Ltm_j is the initial one we obtain the desired map $\mathsf{toh}_j : \mathsf{Ltm}_j \to \mathsf{HEnc}_j$.

```
toh j = rec j (mkNM j (Henc j) envH varH appH
  (λ h' M. lamOf M (λ (m : |M|). eb j h' (m :: M))))
```

Roundtrip at Ltm_j We sketch a conditional proof that $\forall j\, (t : \mathsf{Ltm}_j).\, t \equiv \mathsf{ubd}_j\, (\mathsf{toh}_j t)$, by (nominal) induction, i.e. using the dependent eliminator of Ltm_j. Our proof is conditional since it relies on two axioms. (1) We use a specific axiom called binx that is an instance of standard binary parametricity. It is natural to rely on such an axiom here since our proofs attempt to emulate Kripke *binary* parametricity (the Kripke aspect is in fact emulated by nullary parametricity). An expected model for this standard binary parametricity statement is $\mathsf{Psh}(◪_2 \times □_0 \times □_2)$ where $□_2$ is the original binary affine cube category used by Cavallo and Harper (CH) [CH21] to model binary parametricity, see Section 3.2.4. In fact, we expect binx to be provable in PNTT extended with CH internal binary parametricity operators. (2) Some steps require characterizing the observational parametricity $[-]_0^{\mathcal{O}}$ (defined in Section 3.2.3) of certain terms. These steps are written with a gray background below and are *assumed*. As mentioned above characterizing the observational parametricity $[f]_0^{\mathcal{O}}$ of a complex term f is unwieldy because such characterizations must prove that $[-]_0^{\mathcal{O}}$ commutes with the primitives used in f and reproduce by hand the structure of the term f. We currently have a partial paper proof of these steps. In the future we would like to implement our theory (or maybe an observational version of it) to fully validate this example.

The proofs for constructors other than lam_j are easy. For $t = \mathsf{lam}_j(g : @\mathsf{N} \multimap \mathsf{Ltm}_j)$ we must prove that if the induction hypothesis holds $(g^\bullet : (z : @\mathsf{N}) \multimap (g\, z \equiv \mathsf{ubd}_j\, (\mathsf{toh}_j(g\, z))))$ then $\mathsf{lam}_j\, g \equiv \mathsf{ubd}_j\, (\mathsf{toh}_j\, (\mathsf{lam}_j\, g))$. Let $g^\bullet$ be in context and let us compute the right-hand side RHS of the latter equation. We use the \$ application operator found e.g. in Haskell. This operator is defined as function application but associates to the right, improving clarity.

$$
\begin{aligned}
&RHS \\
&\equiv \mathsf{ubd}_j \,\$\, \lambda M.\, \mathsf{lamOf}\, M \,\$\, \lambda m.\, (\mathsf{ebump}_j \circ (@\mathsf{N} \multimap \mathsf{toh}_j))\, g\, (m :: M) && (\text{def. } \mathsf{toh}_j)\\
&\equiv \mathsf{hlamLtm}_j \,\$\, \lambda m.\, (\mathsf{ebump}_j \circ (@\mathsf{N} \multimap \mathsf{toh}_j))\, g\, (m :: \mathsf{LtmHMod}_j) && (\text{def. } \mathsf{ubd}_j)\\
&\equiv \mathsf{hlamLtm}_j \,\$\, \lambda m.\, (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g\, (m :: \mathsf{LtmHMod}_j) && ([\mathsf{toh}_j]_0^{\mathcal{O}} \equiv \mathsf{toh}_{j+1})\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g\, (\mathsf{var}(\mathsf{c}\, x) :: \mathsf{LtmHMod}_j) && (\text{def. } \mathsf{hlamLtm}_j)\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g \,\$\, \\
&\qquad \mathsf{mkHM}_{j+1}\, \mathsf{Ltm}_j\, (\mathsf{var}(\mathsf{c}\, x) :: \mathsf{env}_j)\, \mathsf{var}_j\, \mathsf{app}_j\, \mathsf{hlamLtm}_j && (\text{def. } ::)
\end{aligned}
$$

The latter model is of type HMod_{j+1} and we observe that it looks similar to $\mathsf{LtmHMod}_{j+1} = \mathsf{mkHM}_{j+1}\, \mathsf{Ltm}_{j+1}\, \mathsf{env}_{j+1}\, \mathsf{var}_{j+1}\, \mathsf{app}_{j+1}\, \mathsf{hlamLtm}_{j+1} : \mathsf{HMod}_{j+1}$. More formally, for $(x : @\mathsf{N})$ in context, the graph of $\mathsf{plug}_x : \mathsf{Ltm}_{j+1} \to \mathsf{Ltm}_j : u \mapsto \mathsf{lbump}_j^{-1}\, u\, x$ turns out be a structure-preserving relation between the two models $\mathsf{LtmHMod}_{j+1}$, $(\mathsf{var}(\mathsf{c}\, x) :: \mathsf{LtmHMod}_j) : \mathsf{HMod}_{j+1}$. This suggests using the binary parametricity of the dependent function $\mathsf{inEnc}_g := (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g : (M : \mathsf{HMod}_{j+1}) \to M$ which appears in our computed RHS. The binary parametricity of inEnc_g is $[\mathsf{inEnc}_g]_2 : \forall (M_0\, M_1 : \mathsf{HMod}_{j+1})(R : [\mathsf{HMod}_{j+1}]_2\, M_0\, M_1).\ |R|(\mathsf{inEnc}_g\, M_0)(\mathsf{inEnc}_g\, M_1)$, where $[\mathsf{HMod}_{j+1}]_2\, M_0\, M_1$ is defined to be the type of structure-preserving relations between M_0, M_1, and $|R| : |M_0| \to |M_1| \to \mathcal{U}$ extracts the carrier of R. The specific binary parametricity axiom we use is $\mathsf{binx} := [\mathsf{inEnc}_g]_2\, \mathsf{LtmHMod}_{j+1}\, (\mathsf{var}(\mathsf{c}\, x) :: \mathsf{LtmHMod}_j)\, (\mathsf{Graph}(\mathsf{plug}_x))$, i.e. the dependent function inEnc_g commutes with the function plug_x. We then have:

$$
\begin{aligned}
RHS &\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g\, (\mathsf{var}(\mathsf{c}\, x) :: \mathsf{LtmHMod}_j)\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, \mathsf{plug}_x \,\$\, (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g\, \mathsf{LtmHMod}_{j+1} && (\mathsf{binx})\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, \mathsf{plug}_x \,\$\, (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g) && (\text{def. } \mathsf{ubd}_{j+1})\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, (\mathsf{lbump}_j^{-1} \$\, (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g))\, x && (\text{def. } \mathsf{plug}_x)
\end{aligned}
$$

Now it remains to show that the latter $\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\, g\, x$, i.e. that the following equality holds $\mathsf{eqNextWorld} : \mathsf{lbump}_j\, g \equiv (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g)$. Let us write $r_j := \mathsf{ubd}_j \circ \mathsf{toh}_j$. (1) The induction hypothesis tells us $g^\bullet : (z : @\mathsf{N}) \multimap (g\, z \equiv r_j(g\, z))$ so by the SAP at $\equiv$ we get $g \equiv \lambda(z : @\mathsf{N}).\, r_j(gz)$ and by applying lbump_j we get $\mathsf{lbump}_j\, g \equiv \mathsf{lbump}_j(\lambda(z : @\mathsf{N}).\, r_j(gz))$. (2) The observational parametricity of r_j is such that

$$[r_j]_0^{\mathcal{O}} \equiv [\mathsf{ubd}_j \circ \mathsf{toh}_j]_0^{\mathcal{O}} \equiv [\mathsf{ubd}_j]_0^{\mathcal{O}} \circ [\mathsf{toh}_j]_0^{\mathcal{O}} \equiv \mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1} = r_{j+1}$$

The function r_j has type $\mathsf{Ltm}_j \to \mathsf{Ltm}_j$ so $[r_j]_0^{\mathcal{O}} \equiv r_{j+1}$ entails (by def. of $[r_j]_0^{\mathcal{O}}$, see Section 3.2.3) $r_{j+1} \circ \mathsf{lbump}_j \equiv \mathsf{lbump}_j \circ (@\mathsf{N} \multimap r_j)$. Applying both sides to $g : @\mathsf{N} \multimap \mathsf{Ltm}_j$ we get $r_{j+1}(\mathsf{lbump}_j\, g) \equiv \mathsf{lbump}_j(\lambda(z : @\mathsf{N}).\, r_j(g\, z))$. (3) Using step (1), (2) and

transitivity of $\equiv$ we get $\mathsf{lbump}_j g \equiv r_{j+1}(\mathsf{lbump}_j\ g) = \mathsf{ubd}_{j+1}\ (\mathsf{toh}_{j+1}\ (\mathsf{lbump}_j\ g))$. Hence eqNextWorld holds and this concludes our proof.

Synthetic Kripke parametricity Kripke parametricity is a strengthened form of binary parametricity that was introduced in a denotational model built by Atkey [Atk09] in order to prove an adequacy result for HOAS: the De Bruijn syntax of ULC is equivalent to the HOAS Π-encoding $\forall A.(A \to A \to A) \to ((A \to A) \to A) \to A$. One way to explain Kripke binary parametricity is to compare it to standard binary parametricity. The latter asserts that polymorphic functions $f : (A : \mathcal{U}) \to T(A)$ map relations R to proofs of relatedness over R (this is explained for $T(A) = A \to A$ in Section 3.1). By contrast, briefly speaking, the *Kripke* binary parametricity of a polymorphic function $f : (A : \mathcal{U}) \to T(A)$ is written $[f]_{\mathsf{K2}}$ and asserts that for every (*) preorder W, f maps any W-monotonic family of relations $(w \mapsto R_w) : W \to A_0 \to A_1 \to \mathcal{U}$ to a monotonic function/proof that for every $w : W$ the outputs $f\ A_0, f\ A_1$ are related over R_w ($\mathcal{U}$ is implicitly ordered by logical implication).

We remark that the example discussed above has semantics closely related to the fragment of Atkey's Kripke model where the W preorder is systematically replaced by $(\mathbb{N}, \leq)$, natural numbers with the usual ordering.[6] Indeed the example above is performed within PNTT with an additional binary parametricity axiom binx. The idea is then that the nullary parametricity aspect of our argument gets translated to the monotonic quantification aspect of Atkey's Kripke model. That is, $(0, 2)$-ary parametricity emulates $(\mathbb{N}, \leq)$-restricted Kripke 2-ary parametricity.

Regarding unrestricted Kripke parametricity, we observe that the quantification (*) on preorders is *hard-coded* into $[f]_{\mathsf{K2}}$, i.e. $[f]_{\mathsf{K2}}$ is not the mere meta-theoretical conjunction of its W-restricted Kripke parametricities. In particular $[f]_{\mathsf{K2}}$ lives one universe level higher than f which is peculiar. Interestingly this hard-coding plays a crucial role in Atkey's proof of the roundtrip at $\forall A.(A \to A \to A) \to ((A \to A) \to A) \to A$. Indeed, within Atkey's model, the proof introduces a variable $B : \mathcal{U}$ and uses $[f]_{\mathsf{K2}}$ at preorder $W := \mathsf{List}\ B$ with prefix ordering, and at type $A_0 := B$ (and some other type for A_1). So B intervenes both to determine the kind of parametricity being used ($W := \mathsf{List}\ B$), and as an input for this parametricity ($A_0 := B$). We see this as a form of diagonalization, which we were not able to internalize.

3.5 Related and Future Work

We have already discussed the most closely related work in internally parametric type theory [BCM15; CH21; VND24] and in nominal type theory [SPG03; Sch06; SS04; Che12; PMD14] in detail in Sections 3.2 and 3.3 respectively. Note that a subtle error

[6] A mismatch: our model uses the category $\square_0$ (Section 3.2.4) while Atkey's uses *preorders* like $(\mathbb{N}, \leq)$.

in the construction of the model shown in [BCM15] was recently spotted in [Alt+24] and investigated in [Reb25].

Regarding the semantics of nominal frameworks, PNTT is modelled (see Section 3.2.4) in $\mathsf{Psh}(\boxslash_2 \times \square_0)$, i.e. bicubical sets over the binary cartesian cube category $\boxslash_2$ and the nullary affine cube category $\square_0$. The latter is also the base category of the Schanuel sheaf topos [Pit13, §6.3], which is in turn equivalent to the category of nominal sets [Pit13], used to model FreshMLTT [PMD14]. FreshML [SPG03] is modelled in the Fraenkel-Mostowski model of set theory, which inspired the development of nominal sets. Schöpp and Stark's bunched system [SS04] is modelled in a class of categories, including the Schanuel topos, that are locally cartesian closed and also equipped with a semicartesian closed structure. Finally, $\lambda^{\Pi\mathcal{U}}$ [Che12] has a syntactic soundness proof.

$\square_0$ is a category of cubes without diagonals, so the internal language of its associated presheaf topos is among the first candidates to get a transpension type [ND24] with workable typing rules [Nuy25]. Dual to the fact that universal and existential name quantification semantically coincide, so do freshness and transpension. As such, the transpension type is already present as Gel in the current system, but Gel's typing rules are presently weaker: the rules GelF and GelI remove a part of the context, rather than quantifying it. The relationship between Gel and transpension is explained in more generality in [ND24].

Regarding the implementation of PNTT, given that the binary CH theory was implemented in [VND24] on top of Cubical Agda [VMA21], implementing PNTT (essentially nullary CH + name induction) is feasible. We think that such an implementation would be usable but not particularly scalable by default to perform parametricity proofs. The issue is that such proofs require the SAP, which cannot be proved as a single internal theorem (see Section 3.4.2 and [VND24]). A practical solution could be to automate the process of building SAP proofs. Alternatively, one could instead build PNTT on top of *observational* [Alt+24] nullary PTT, where the SAP is not needed since a first-class recursive translation attempts to replace the Bridge type. The Narya proof assistant in progress [Nar25b] implements observational n-ary PTT (and more).

Chapter 4

Adequacy of (P)HOAS by binary parametricity

In this short chapter we transcribe in Agda --bridges a proof that was communicated to us by Andrea Vezzosi, and expand on it[1].

Vezzosi's proof is concerned with a representation of syntax called Parametric Higher-Order Abstract Syntax (PHOAS, [Chl08]), defined in the next section. His proof can be understood as an adequacy result for PHOAS: it shows that the type of closed PHOAS untyped λ-terms injects into the more familiar and standard type of closed De Bruijn untyped λ-terms. The proof is performed using (regular) binary parametricity. The key idea is that scoping information can be represented using parametricity. Originally the proof was formalized in the experimental `parametric` branch of Agda, an implementation of the type theory described in [NVD17].

Our own contribution in this chapter is as follows:

- All the results in the present list are formalized in the Agda --bridges proof assistant, introduced in Chapter 2.
- We show an equivalence (vs. injection above) between the PHOAS and the De Bruijn representation.
- The equivalence holds for open terms that may contain j free variables, for all j (vs. $j = 0$ above). Making the proof work for open terms requires using De Bruijn *levels* in a certain intermediary syntax representation (contrary to De Bruijn indices, De Bruijn levels read the context/binders from left to right).
- We make the simple observation that Atkey already proved an equivalence between PHOAS and HOAS (for $j = 0$, see [Atk09], Theorem 2). His proof uses regular binary parametricity. We extend the proof to all j.

[1] See `https://github.com/antoinevanmuylder/bridgy-lib/tree/main/Bridgy/Examples/`

- Therefore this yields a proof of adequacy for HOAS by regular binary parametricity.

The latter fact was surprising to us. Indeed in [Atk09], Atkey provides another proof of adequacy for HOAS: one that uses *Kripke* parametricity (briefly evoked in Chapter 3). Kripke parametricity is much stronger than standard parametricity, and Atkey's proof crucially relies on this additional strength.

The rest of this chapter is organized as follows. Section 4.1 provides basic definitions. Section 4.2 proves the adequacy of PHOAS, based on Vezzosi's proof. The key idea of the proof emerges in Section 4.2.1 where a map from PHOAS to De Bruijn is defined by parametricity. The rest of the proof is sketched. Lastly in Section 4.3 we recall Atkey's proof that PHOAS and HOAS are equivalent.

4.1 Basic Definitions

First, Fin j is the finite type with j elements referred to in the code as zero, one, two, three, ...

```
data Fin : ℕ → Type where
  zero : {j : ℕ} → Fin (suc j)
  suc  : {j : ℕ} (i : Fin j) → Fin (suc j)
```

Definition 5 (Scoped De Bruijn for ULC)**.** The De Bruijn representation of the untyped λ-calculus is given by the following data type family. The family is indexed by context sizes $j : \mathbb{N}$ so the representation is said to be scoped. A simple example of a term follows the definition.

```
data Lam (j : ℕ) : Type where
  var : Fin j → Lam j
  app : Lam j → Lam j → Lam j
  lam : Lam (suc j) → Lam j

-- x, y ⊢ λ z. z x
example : Lam 2
example = lam (app (var zero) (var two))
```

The idea behind this representation is that var n refers to the n-th variable (from right to left) introduced by binders or by the context, where we start the count with $n = 0$.

As a representation of syntax with binders, the De Bruijn representation offers a unique set of tradeoffs which we list here. (1) It is a first-order representation since no constructors admit metatheoretic (i.e. Agda) functions as arguments. (2) Defining and establishing properties about the representation can be done in regular, plain type theory (with enough support for data types). (3) Terms that are α-

equivalent when written on paper, i.e. equal up to consistent renaming of bound variables, are automatically equal in the representation. Syntax representations that do not enforce this property are hard to manipulate since functions f defined on the representation must often be accompanied by a proof that f respects α-equivalence. (4) The representation possesses an induction principle, i.e. a way to define (dependent) functions/proofs out of the representation. (5) However it can be argued that representing variables with natural numbers is not intuitive. (6) More annoyingly, weakening is not structural but implemented via an explicit index-shifting operation. As a consequence, definitions and proofs over the syntax typically require auxiliary lemmas establishing that the constructions under consideration commute with, or are stable under, shifting. (7) Another peculiarity is that Lam 0 can not be defined in isolation since its definition mentions Lam 1, and so on.

Definition 6 (PHOAS for ULC)**.** The PHOAS representation/encoding of the untyped λ-calculus is given by the type family PhoasEnc below. The family is indexed by context sizes $j : \mathbb{N}$ so the representation is said to be scoped. An example follows the definition.

```
data PhoasD (V : Type) : Type where
  var : V → PhoasD V
  app : PhoasD V → PhoasD V → PhoasD V
  lam : (V → PhoasD V) → PhoasD V

PhoasEnc : ℕ → Type₁
PhoasEnc j = ∀ (V : Type) → (Fin j → V) → PhoasD V

-- x, y ⊢ λ z. z x
example : PhoasEnc 2
example = λ V env → lam (λ z → app (var z) (env one))
```

The PHOAS representation offers its own set of tradeoffs. (1) It is said to be higher-order because the lam constructor is a second-order function, i.e. it takes a metatheoretic (i.e. Agda) function $V \to \mathsf{PhoasD}\ V$ as an argument. (2) α-equivalent PHOAS terms are equal, because Agda functions $V \to \mathsf{PhoasD}\ V$ respect this property. (3) The PHOAS representation can be defined and used in plain type theory and possesses, to some extent, an induction principle. Indeed using PHOAS in practice means working in a context where a type of variables V : Type is postulated (this can be seen as a drawback). Then the induction principle of PhoasD V can be used. (4) The issues related to weakening from the De Bruijn representation do not arise when using PHOAS. Indeed weakening a PHOAS term is simply done by postulating an extra $v : V$ in the Agda context, and Agda functions and proofs are automatically stable under such weakenings. (5) PhoasEnc 0 can be defined independently from other types of the family. (6) PhoasEnc lives one universe level higher than Lam since it is obtained as a quantification on all types V.

One conceptual drawback of the PHOAS representation is that stating and proving

its adequacy is subtle. First, it is not the case that if an abstract type of variables V is postulated, the type PhoasD V is equivalent to e.g. Lam 0 from Definition 5. More precisely this fact can not be proved in Agda because there exist concrete V's for which it fails. Indeed PhoasD $\mathbb{N}$ contains exotic terms, i.e. terms that correspond to no actual λ-term. This is due to the fact that the lam constructor expects an argument $\mathbb{N} \to$ PhoasD $\mathbb{N}$ and one can provide such an argument by pattern-matching on the domain $\mathbb{N}$. Second, hence the correct way to state adequacy in Agda is to assert an equivalence between the De Bruijn representation Lam j on one side, and on the other side the Π-type of PHOAS polymorphic programs PhoasEnc $j = (V :$ Type$) \to ($Fin $j \to V) \to$ PhoasD V. However providing such an equivalence requires parametricity. This is discussed in the next section.

Definition 7 (Scoped HOAS for ULC)**.** The HOAS representation/encoding of the untyped λ-calculus is given by the type family HoasEnc below. The family is indexed by context sizes j so the representation is said to be scoped. An example follows the definition.

```
record HoasMod (j : ℕ) : Type₁ where
  constructor mkHoasMod
  field
    Cr : Type
    envOf : Fin j → Cr
    appOf : Cr → Cr → Cr
    lamOf : (Cr → Cr) → Cr

-- ∀ (H : Type) → (Fin j → H) → (H → H → H) → ((H → H) → H) → H
HoasEnc : ℕ → Type₁
HoasEnc j = ∀ (M : HoasMod j) → M .Cr -- uncurried

-- x, y ⊢ λ z. z x
example : HoasEnc 2
example = λ M → M .lamOf (λ z → M .appOf z (M .envOf one))
```

Note that terms k : HoasEnc j are uncurried to ease the process of calling their parametricity using the ROTT DSL (this is discussed in Section 4.3).

The tradeoffs offered by HOAS are as follows. (1) It is a higher-order representation since the lam "constructor" takes an Agda function $H \to H$ as input. (2) α-equivalent HOAS terms are equal because Agda functions $H \to H$ respect this property. (3) The representation can be defined in plain type theory. (4) The issues related to weakening from the De Bruijn representation do not arise when using HOAS. Furthermore, substitution of HOAS λ-terms has an elementary definition: it is simply (Agda) function application. (5) HoasEnc 0 can be defined independently from other types of the family. (6) Annoyingly the representation lacks any kind of induction principle. (7) HoasEnc lives one universe level higher than Lam since it is obtained as a quantification on all types M.

Drawback (6) is quite limiting and is alleviated in some systems (e.g. Haskell) by rather defining the HOAS representation as the following data type (when $j = 0$).

```
HoasD : Type where
  app : HoasD → HoasD → HoasD
  lam : (HoasD → HoasD) → HoasD

foo : HoasD
foo = lam (λ x → λ { app a b → app a b ; lam f → f x})
```

However this solution is far from perfect. Firstly, this data type declaration is not valid in Agda because the system recognizes the leftmost occurrence of HoasD in the type of lam as problematic. The issue is that the type being defined HoasD occurs directly to the left of an arrow in a constructor argument type, i.e., this occurrence is not *strictly positive*. This makes it possible to write programs that don't normalize such as g foo where g denotes the argument of the foo program above, i.e., foo $=$ lam g (the $\lambda\{\}$ syntax is used for anonymous pattern matching). And secondly, HoasD is not equivalent to the De Bruijn representation because of the presence of exotic terms such as foo.

4.2 Adequacy of PHOAS

We now prove that Lam $j \simeq$ PhoasEnc j for all j.

4.2.1 Mapping out of PHOAS

The most difficult part conceptually is to devise a map fromPhoas : PhoasEnc $j \to$ Lam j. We define the map and provide some insights about the definition afterwards.

In order to define the map we need the unary parametricity translation of PhoasD : $(V$: Type$) \to$ Type. Technically since PhoasD V is a data type, its unary translation denoted $[\text{PhoasD}]'_1$ below is itself a data type, indexed once.

```
data [PhoasD]₁' (V : Type) (Vdot : V → Type) : PhoasD V → Type where
  vardot : ∀ v → Vdot v → [PhoasD]₁' V Vdot (var v)
  appdot : ∀ a b →
    [PhoasD]₁' V Vdot a →  [PhoasD]₁' V Vdot b →
    [PhoasD]₁' V Vdot (app a b)
  lamdot : ∀ (f : V → PhoasD V) →
    (fdot : ∀ v → Vdot v → [PhoasD]₁' V Vdot (f v)) →
    [PhoasD]₁' V Vdot (lam f)
```

Yet the above definition of $[\text{PhoasD}]'_1$: PhoasD $V \to$ Type turns out to be equivalent to the following simpler definition, by recursion on the domain PhoasD V. The proof

of the equivalence is straightforward and proceeds by induction on PhoasD V in one direction, and on $[\mathsf{PhoasD}]'_1$ in the other.

Definition 8 (Unary translation of PhoasD)**.** The unary parametricity translation of the PhoasD V data type is (equivalent to) the following recursively defined family.

```
[PhoasD]₁ : (V : Type) (Vdot : V → Type) → PhoasD V → Type
[PhoasD]₁ V Vdot (var v) = Vdot v
[PhoasD]₁ V Vdot (app a b) = [PhoasD]₁ V Vdot a × [PhoasD]₁ V Vdot b
[PhoasD]₁ V Vdot (lam f) = ∀ v → Vdot v → [PhoasD]₁ V Vdot (f v)
```

Next, we define the expected map $\mathsf{PhoasEnc}\, j \to \mathsf{Lam}\, j$ in two steps[2] called descope and rescope.

$$\mathsf{PhoasEnc}\, j \xrightarrow{\text{descope}} \sum_{d:\mathsf{PhoasD}\,\mathbb{N}} \forall\, n.\, [\mathsf{PhoasD}]_1\, \mathbb{N}\, (\lambda x.\, x < n + j)\, d \xrightarrow{\text{rescope}} \mathsf{Lam}\, j$$

The central type in the diagram is an intermediate syntax representation possessing some unusual features. Its first feature can not be seen in the type but is rather a convention. Contrary to the two other representations above, we adopt the convention that terms of $\mathsf{PhoasD}\,\mathbb{N}$ use De Bruijn *levels* instead of indices to refer to free variables. The difference with respect to indices is that $\mathsf{var}\, n : \mathsf{PhoasD}\,\mathbb{N}$ refers to the n-th variable bound by the context, where we count *from left to right* and start the count with 0. For example, for $j = 2$, the λ-term $x, y \vdash \lambda z.\, y\, z$ is written `lam (λ z → app (var 1) (var z))` in Agda. Adopting this convention is useful to define the rescope map because consistently rescoping a body b can be done by computing $b\, j$ (using De Bruijn indices, one would like to compute $b\, 0$ but it clashes with the last free variable in the context). Second, the intermediate representation separates terms $d : \mathsf{PhoasD}\,\mathbb{N}$ and proofs that they are well-scoped. Vezzosi's insight is: (1) to express scoping information about $d : \mathsf{PhoasD}\,\mathbb{N}$ using $[\mathsf{PhoasD}]_1$. If $d = \mathsf{var}\, v$, $[\mathsf{PhoasD}]_1\, \mathbb{N}\, (\lambda x.\, x < j)\, d$ computes to $v < j$. And (2) to assert not only $[\mathsf{PhoasD}]_1\, \mathbb{N}\, (\lambda x.\, x < j)\, d$ but also $[\mathsf{PhoasD}]_1\, \mathbb{N}\, (\lambda x.\, x < n + j)\, d$ for all n. The latter quantification is crucial to define the second step rescope for lam values.

4.2.1.1 Rescoping

The rescope map is defined as follows. The types of some auxiliary functions are provided below.

```
rescope : ∀ {j} → (d : PhoasD ℕ)
      (hyp : ∀ n → [PhoasD]₁ ℕ (λ x → x < (n + j)) d) → Lam j
rescope {j} (var v) hyp =
  var (<ToFin _ (j ∸ (suc v) , ∸-compl v j (hyp 0)))
rescope {j} (app a b) hyp =
```

[2] Not unlike the HOAS adequacy proof by Kripke parametricity from [Atk09].

```
    app (rescope a (λ n → hyp n .fst)) (rescope b (λ n → hyp n .snd))
rescope {j} (lam f) hyp =
  lam (rescope {suc j} (f j) λ n →
  -- goal: [PhoasD]₁ ℕ (λ x → x < (n + suc j)) (f j)
  subst (λ blk → [PhoasD]₁ ℕ (λ x → x < blk) (f j)) (sym (+-suc n j)) -- just a subst
  -- goal: [PhoasD]₁ ℕ (λ x → x < suc (n + j)) (f j)
  (hyp (suc n) j (j<sucn+j j n)))

<ToFin : ∀ j → (Σ[ v ∈ ℕ ] v < j) → Fin j
_∸_ : ℕ → ℕ → ℕ -- truncated subtraction
∸-compl : ∀ v j → v < j → (j ∸ (suc v)) < j
+-suc : ∀ m n → m + suc n ≡ suc (m + n)
j<sucn+j : ∀ j n → j < suc (n + j)
```

As can be seen in the var clause of rescope, the intermediate representation uses De Bruijn levels and a level v translates to an index $j \dot{-} (\mathsf{suc}\, v)$. The $\dot{-}$ function is the usual subtraction, but outputs 0 when the second argument is too large. Regarding the lam clause, rescoping $\mathsf{lam}\, f : \mathsf{PhoasD}\, \mathbb{N}$ involves computing $f\, j$, and evaluating the scoping hypothesis one step further $\mathsf{hyp}\, (\mathsf{suc}\, n)$.

4.2.1.2 Descoping

The first component of the descope map sends a PHOAS term $k : \mathsf{PhoasEnc}\, j$ to the term $k\, \mathbb{N}\, (\mathsf{rev}\delta\mathbb{N}\, j) : \mathsf{PhoasD}\, \mathbb{N}$ where $\mathsf{rev}\delta\mathbb{N} : \forall j.\, \mathsf{Fin}\, j \to \mathbb{N}$ maps $\{0, 1, \ldots, ..., j - 1\}$ to $\{j-1, j-2, \ldots, ..., 0\}$. The reversal accounts for the fact that the intermediary representation uses De Bruijn levels for free variables by contrast with $\mathsf{PhoasEnc}\, j$ which uses indices.

In a system featuring internal *unary* parametricity, the second component would be straightforward to define. Indeed the second component has type $\mathsf{descope}_2\, k : \forall n.\, [\mathsf{PhoasD}]_1\, \mathbb{N}\, (\lambda x.\, x < n + j)\, (k\, \mathbb{N}\, (\mathsf{rev}\delta\mathbb{N}\, j))$. And k and its unary parametricity would have the following types.

```
k : (V : Type) → (Fin j → V) → PhoasD V
[k]₁ : (V : Type) (Vdot : V → Type) (env : Fin j → V) (envdot : ∀ x → Vdot (env x)) →
  [PhoasD]₁ V Vdot (k V env)
```

So one can define $\mathsf{descope}_2\, k = \lambda n.\, [k]_1\, \mathbb{N}\, (\lambda x.\, x < n + j)\, (\mathsf{rev}\delta\mathbb{N}\, j)\, \mathsf{pf}$ where $\mathsf{pf} : \forall x.\, \mathsf{rev}\delta\mathbb{N}\, j\, x < n + j$ is trivial. As a side note, technically the type of $[k]_1$ is merely equivalent to the above type. For instance, the actual translation of PhoasD is rather $[\mathsf{PhoasD}]'_1$ evoked above. Additionally note that the translation of the data type $\mathsf{Fin}\, j$ is $\lambda v.\, \mathsf{Unit} : \mathsf{Fin}\, j \to \mathsf{Type}$, i.e. the $\mathsf{Fin}\, j$ type is bridge-discrete as is often the case with data types.

Agda --bridges does not feature internal unary parametricity, but rather internal *binary* parametricity. Fortunately it turns out that it is possible to emulate the above instance of unary parametricity by using binary parametricity instead. This is the

content of the emulate function below. The types of some auxiliary functions are also provided.

```
[PhoasD]₂ : (V0 V1 : Type) (R : V0 → V1 → Type) (t0 : PhoasD V0)
    (t1 : PhoasD V1) → Type
[PhoasD]₂ V0 V1 R (var x0) (var x1) = R x0 x1
[PhoasD]₂ V0 V1 R (app a0 b0) (app a1 b1) =
  ([PhoasD]₂ V0 V1 R a0 a1) × ([PhoasD]₂ V0 V1 R b0 b1)
[PhoasD]₂ V0 V1 R (lam f0) (lam f1) =
  ∀ x0 x1 → R x0 x1 → [PhoasD]₂ V0 V1 R (f0 x0) (f1 x1)
[PhoasD]₂ V0 V1 R _ _ = ⊥

emulate : ∀ j n → (df : PhoasD (Fin (n + j))) (dn : PhoasD ℕ) →
  [PhoasD]₂ (Fin (n + j)) ℕ (λ y n → toℕ y ≡ n) df dn →
  [PhoasD]₁ ℕ (λ x → x < (n + j)) dn
emulate j n (var x0) (var x1) rel =
  subst (λ blk → blk < (n + j)) rel (toℕRemainsLt (n + j) x0)
emulate j n (app a0 b0) (app a1 b1) rel =
  emulate j n a0 a1 (rel .fst) , emulate j n b0 b1 (rel .snd)
emulate j n (lam f0) (lam f1) rel = λ v v<n+j →
  emulate j n (f0 (<ToFin _ (v , v<n+j))) (f1 v)
  (rel (<ToFin _ (v , v<n+j)) v (toℕ-<ToFin _ v v<n+j))

toℕ : ∀ {j} → Fin j → ℕ
toℕRemainsLt : ∀ j (k : Fin j) → toℕ k < j
toℕ-<ToFin : ∀ j k (hypk : k < j) → toℕ (<ToFin j (k , hypk)) ≡ k
```

All in all descope$_2$ can be defined by binary parametricity. Within its definition, the appropriate invocation of internal binary parametricity is shown in the auxparam program below.

```
descope₂ : ∀ j → (k : PhoasEnc j) →
      ∀ n → [PhoasD]₁ ℕ (λ x → x < (n + j)) (k ℕ (revδℕ j))
descope₂ j k n = emulate j n
  (k (Fin (n + j)) (revδF n j)) (k ℕ (revδℕ j)) (auxparam j n k)

revδF n j : Fin j → Fin (n + j) -- outputs j-1, ... , 0 : Fin (n + j)

auxparam : ∀ j n (k : PhoasEnc j) →
  [PhoasD]₂ (Fin (n + j)) ℕ (λ y z → toℕ y ≡ z)
  (k (Fin (n + j)) (revδF n j)) (k ℕ (revδℕ j))
auxparam = param (TypeRRG ℓ-zero) (PhoasEncDRRG j) k
  (Fin (n + j)) ℕ (λ y z → toℕ y ≡ z)
  (revδF n j) (revδℕ j) relatedEnvs
  where
    relatedEnvs : ∀ x₀ x₁ → x₀ ≡ x₁ → toℕ (revδF n j x₀) ≡ revδℕ j x₁
```

The auxparam lemma is a consequence of the parametricity of the dependent function $k : \forall V.\, (\mathsf{Fin}\, j \to V) \to \mathsf{PhoasD}\, V$. Intuitively the parametricity of k asserts that k maps any relation $R : V_0 \to V_1 \to \mathsf{Type}$ to a proof of structural relatedness over R, i.e., a proof that if input environments are related over R then k's outputs are related over R. Here the relation R is set to be the graph of the function $\mathsf{toℕ} : \mathsf{Fin}(n + j) \to \mathbb{N}$. More formally, the proof of auxparam is performed using the ROTT DSL/library discussed in Chapter 2. Indeed the internal parametricity of k is invoked with the param function of the ROTT library. In particular k's dependent codomain $V \vDash (\mathsf{Fin}\, j \to V) \to \mathsf{PhoasD}\, V$ is expressed using the ROTT library (PhoasEncDRRG in the code) and provided to the param function. The remaining arguments of param contain the relation R and a proof that both supplied environments are related over R.

4.2.2 Rest of the proof

Defining a map from the De Bruijn representation to the PHOAS representation is straightforward.

```
toPhoas : ∀ j → Lam j → ∀ V → (Fin j → V) → PhoasD V
toPhoas j (var x) = λ V env → var (env x)
toPhoas j (app a b) = λ V env → app (toPhoas j a V env) (toPhoas j b V env)
toPhoas j (lam b) = λ V env → lam λ v → toPhoas (suc j) b V (v :: env)
  where
    _::_ : ∀ {V j} → V → (Fin j → V) → Fin (suc j) → V
    _::_ v env zero = v
    _::_ v env (suc x) = env x
```

The roundtrip equality at De Bruijn $\forall(t : \mathsf{Lam}\, j).\, \mathsf{fromPhoas} j (\mathsf{toPhoas}\, j\, t) \equiv t$ is proved by induction on $(t : \mathsf{Lam}\, j)$, without using additional parametricity lemmas. The other roundtrip equality $\forall(t : \mathsf{PhoasEnc}\, j).\, \mathsf{toPhoas}\, j (\mathsf{fromPhoas}\, j\, t) \equiv t$ requires using the binary parametricity of t, in a way similar to how it is used to define the descope map.

4.3 PHOAS and HOAS are equivalent

Finally we show that $\mathsf{PhoasEnc}\, j \simeq \mathsf{HoasEnc}\, j$, following [Atk09], Theorem 2. A slight difference with the latter is that our PHOAS encoding uses the PhoasD data type whereas this data type is rather expressed as a Church encoding in [Atk09].

Yet the idea of the proof remains the same. Defining maps in and out of HOAS does not require parametricity, and both roundtrips do require it. The maps from and to HOAS are defined as follows.

```
toHoas' : ∀ {j} → (M : HoasMod j) → PhoasD (M .Cr) → M .Cr
toHoas' M (var x) = x
```

```
toHoas' M (app a b) = M .appOf (toHoas' M a) (toHoas' M b)
toHoas' M (lam f) = M .lamOf λ x → toHoas' M (f x)

toHoas : ∀ {j} → (∀ V → (Fin j → V) → PhoasD V) → HoasEnc j
toHoas k M = toHoas' M (k (M .Cr) (M .envOf))

fromHoas : ∀ {j} → HoasEnc j → (∀ V → (Fin j → V) → PhoasD V)
fromHoas k V env = k (mkHoasMod
  (PhoasD V)
  (λ x → var (env x))
  app
  λ f → lam λ v → f (var v))
```

In order to prove the roundtrip equality for a PHOAS term k, the basic idea is to use the parametricity of k at the relation R given by the (inverse of the) graph of the var : $V \to$ PhoasD V function. As explained above, the parametricity of k is invoked with the param function of the ROTT library. The remaining arguments contain the relation R and a proof that both supplied environments are related over R.

```
Phoas≤ : ∀ {j} → (k : ∀ V → (Fin j → V) → PhoasD V) → fromHoas (toHoas k) ≡ k
Phoas≤ {j} k = funExt λ V → funExt λ env →
  Phoas≤aux (k (PhoasD V) (λ x → var (env x))) (k V env) -- see lemma below
    (param (TypeRRG ℓ-zero) (PhoasEncDRRG j) k
    (PhoasD V) V (λ d v → d ≡ var v)
    (λ x → var (env x)) env
    (λ x0 x1 x2 → cong (λ x → var (env x)) x2))

Phoas≤aux : ∀ {V : Type} {j : ℕ} {env : Fin j → V} →
  ∀ d0 d1 (d2 : [PhoasD]₂ (PhoasD V) V (λ d v → d ≡ var v) d0 d1) →
    toHoas' (mkHoasMod (PhoasD V) (λ x → var (env x))
    app (λ f → lam (λ v → f (var v)))) d0
    ≡ d1
Phoas≤aux (var d0) (var x) d2 = d2
Phoas≤aux (app a0 b0) (app a1 b1) d2 =
  cong₂ app (Phoas≤aux a0 a1 (d2 .fst)) (Phoas≤aux b0 b1 (d2 .snd))
Phoas≤aux (lam f0) (lam f1) d2 = cong lam (funExt λ v →
  Phoas≤aux (f0 (var v)) (f1 v) (d2 (var v) v refl))
```

For the other roundtrip, given a term k : HoasEnc j, we also wish to use the parametricity of k at a certain relation. This is again done using the ROTT DSL, that is to say, one has to express k's domain HoasMod j and dependent codomain M : HoasMod $j \vDash M.Cr$ using the primitives of ROTT. This codomain is written carrierdRRG in the code. Next the parametricity of k expects a binary relation R as input, and a proof that it respects the structure of HoasMod j. Given a model M : HoasMod j, we pick $R : M$ .Cr $\to$ PhoasD (M .Cr) $\to$ Type, $R = \lambda\, m\, d.$ toHoas$'$ $M\, d \equiv m$. The proof that R relates each field of both models is the last argument of param below.

```
Hoas≤ : ∀ {j} → (k : HoasEnc j) → toHoas (fromHoas k) ≡ k
Hoas≤ {j} k = funExt λ M →
  param (HoasModRRG j) carrier-dRRG k
  M
  (mkHoasMod (PhoasD (M .Cr)) (λ x → var (M .envOf x))
    app (λ f → lam (λ v → f (var v))))
  -- R with the proof that it respects the structure of HoasMod
  ([mkHoasMod]₂ (λ m d → toHoas' M d ≡ m)
      (λ x0 x1 x2 → cong (M .envOf) (sym x2))
      (λ a0 a1 a2 b0 b1 b2 → cong₂ (M .appOf) a2 b2)
      λ f0 f1 f2 → cong (M .lamOf) (funExt λ m → f2 m (var m) refl))

-- The type of structural relations between (M0 M1 : HoasMod j)
record [HoasMod]₂  (j : ℕ) (M0 M1 : HoasMod j) : Type₁ where
  constructor [mkHoasMod]₂
  field
    [Cr]₂ : M0 .Cr → M1 .Cr → Type
    [envOf]₂ : ∀ (x0 x1 : Fin j) → x0 ≡ x1 → [Cr]₂ (M0 .envOf x0) (M1 .envOf x1)
    [appOf]₂ : (a0 : M0 .Cr) (a1 : M1 .Cr) (a2 : [Cr]₂ a0 a1) →
      (b0 : M0 .Cr) (b1 : M1 .Cr) (b2 : [Cr]₂ b0 b1) →
      [Cr]₂ (M0 .appOf a0 b0) (M1 .appOf a1 b1)
    [lamOf]₂ : (f0 : M0 .Cr → M0 .Cr) (f1 : M1 .Cr → M1 .Cr)
      (f2 : ∀ y0 y1 (y2 : [Cr]₂ y0 y1) → [Cr]₂ (f0 y0) (f1 y1)) →
      [Cr]₂ (M0 .lamOf f0) (M1 .lamOf f1)
```

Chapter 5

Conclusion

5.1 Global Discussion of Research Results

This thesis makes contributions to binary and nullary internal parametricity. Chapter 2, Chapter 4 are concerned with the tooling, methodology and consequences of binary internal parametricity. Chapter 3 concerns a new foundational connection between nullary internal parametricity and nominal type theory.

5.1.1 Binary Internal Parametricity: Agda --bridges and ROTT

Parametricity translations are convenient because they are computational in nature. They compute both the statement and the proof of a free theorem directly from a program. But translations are limited to individual programs: they cannot validate global free theorems, which quantify internally over all programs of a given polymorphic type. Internally parametric type theories do validate such global statements, but at the time this thesis began parametricity proofs were typically performed (1) on paper and (2) in a low-level fashion.

Agda --bridges addresses the first gap by providing a full proof-assistant implementation of the CH theory, adapted to the cubical type theory (CCHM) underlying Agda --cubical [VMA21]. The implementation is non-trivial: First, the substructurality of the CH theory, i.e. the requirement that bridge variables be fresh with respect to the rest of the context, is realized through freshness typechecking constraints, building on the implementation of ticks from Agda --guarded [VV20; VV23]. Second, the Kan operations hcomp and transp of Agda --cubical are re-implemented against an extended cofibration logic that accommodates bridge faces alongside path faces. Agda --bridges successfully typechecks the entire Agda --cubical standard library [Agd], so results from HoTT like univalence are available in Agda --

bridges. Several theorems previously proved only on paper in the CH theory are given their first fully machine-checked proofs here, the most notable being *relativity* [CH21]: $(A_0 \to A_1 \to \mathsf{Type}) \simeq \mathsf{Bridge}\ \mathsf{Type}\ A_0\ A_1$, the relational counterpart of univalence.

ROTT addresses the second gap, namely usability. The core observation is that the effort required to derive a free theorem from bridge primitives is dominated by the need to establish, for each type former appearing in the target type, a commutation law between the Bridge type former and that type former. These laws collectively constitute the Structure Relatedness Principle (SRP), the relational analogue from the Structure Identity (SIP) from HoTT/UF [Uni13]. Once the SRP has been established for a type, the general parametricity theorem param, an internal and observational abstraction theorem, immediately yields the desired free theorem as a one-liner.

The remaining challenge is to prove the SRP itself efficiently. Several tools that simplify SIP proofs do not carry over to the relational setting: the fundamental theorem of identity types has no straightforward relational counterpart, and proof-irrelevant components of a type cannot simply be ignored when establishing the SRP, contrary to the SIP. ROTT circumvents this by providing a shallowly embedded type theory (basically a category with families (CwF) internal to Agda --bridges) whose objects are types equipped with a proof of the SRP (called relativistic reflexive graphs, or RRGs) and whose type and term formers are closed under the SRP by construction. Expressing a target type in the ROTT language is therefore both necessary and sufficient to establish the SRP, after which param can be applied.

Agda --bridges has been used to formalize free theorems of genuine interest, including Chapter 4.

5.1.2 Nullary Internal Parametricity: Nominal Type Theory

Chapter 3 is concerned with a different application of internal parametricity, at arity zero. The starting observation is that the universal name abstraction type former $@N \multimap -$ of nominal frameworks satisfies precisely the same rules as the nullary bridge type of nullary PTT: both are affine function types. This was also observed in [Nar25b]. This identification is the starting point behind Parametric Nominal Type Theory (PNTT).

On its own, nullary PTT already supplies several nominal features: universal name abstractions as bridge types, typal freshness expressed via Gel types, non-primitive existential name abstractions derivable from Gel, a name-swapping operation, and the local-scoping primitive ν of [PMD14]. The Gel type provides a clean account of typal freshness and proves $\mathcal{U} \simeq (@N \multimap \mathcal{U})$, which had not previously appeared in the nominal literature to our knowledge.

However, recovering the full strength of nominal frameworks requires a first-class type of names $\vdash \mathsf{Nm}$. The novel contribution is *name induction*: given a term $n : \mathsf{Nm}$ and a bridge variable x, name induction performs a case split yielding either $n = x$ or x is fresh in n. This principle is what ultimately enables nominal pattern matching

for nominal syntax. Previous nominal frameworks axiomatizing $@N \multimap -$ as fresh functions could only afford a weak form of pattern matching for nominal data types where values of $@N \multimap -$ can not be inspected.

The central technical result is the *Structure Abstraction Principle* (SAP), the nullary counterpart of the SRP. The SAP asserts that $@N \multimap -$ commutes with every type former, particularly Nm itself. For standard type formers the SAP follows from ext and Gel, as in the binary case. For Nm, the SAP reads $1 + \mathsf{Nm} \simeq (@N \multimap \mathsf{Nm})$ and is proved using name induction. The SAP at Nm is the key that unlocks missing operations: the previously undefinable free names function $\mathsf{fns} : \mathsf{Ltm} \to \mathsf{List}\,\mathsf{Nm}$ can now be completed because $@N \multimap \mathsf{List}\,\mathsf{Nm}$ reduces via the SAP to $\mathsf{List}(@N \multimap \mathsf{Nm}) \simeq \mathsf{List}(1 + \mathsf{Nm})$, and the missing clause is then defined by filtering out the left summand.

5.2 Implications of the Research

Towards a mainstream parametricity toolkit Prior to this thesis, practitioners wishing to use parametricity in a proof assistant had to choose between external translations (no global theorems) and internally parametric systems with no usable implementation. Agda --bridges and ROTT together constitute the first practical answer to this dilemma: global free theorems are provable, and the proof overhead is controlled. This lowers the barrier to using parametricity as a routine proof technique rather than a specialized research tool.

Univalence and relativity as complementary extensionality principles Agda --bridges validates, coming from the CH theory, that the SIP (equality is isomorphism) and the SRP can coexist in the same system without conflict. This confirms a picture in which univalence and parametricity are two independent but compatible extensionality principles, suggesting that future proof assistants may provide both as first-class features.

A parametric foundation for nominal reasoning PNTT proposes that internally parametric type theory is expressive enough to serve as a foundation for nominal dependent type theory. This seems like a fruitful connection which could be further explored.

Verified metatheory of programming languages Applications of nominal syntax are prominent in the mechanized metatheory of programming languages and proof systems: substitution lemmas, subject reduction, normalisation arguments, and so on. PNTT offers a setting in which such reasoning can be carried out with a convenient representation (α-equivalence for free, human-readable names).

5.3 Future Perspectives

Observational PTT There is a gap between the bridge type of a type and its translation, namely the SRP/SAP. In the present system the SRP and SAP hold only propositionally and must be proved on a case-by-case basis for each type former, then composed for composite types. Observational PTT attempts to turn the translation for types and terms, not the Bridge type, into a first-class operation. Existing such systems partially reach this goal, and either have laborious sets of rules that are not proved sound [BM12] or provide span-based parametricity [Alt+24] instead of usual, indexed parametricity. Narya [Nar25a] is a work-in-progress implementation that has observational indexed parametricity and builds on the work in [Alt+24].

An implementation of Nominal Type Theory Given that Agda --bridges implements the binary CH theory, implementing PNTT from Chapter 3 is feasible, since the latter is essentially the nullary CH theory + a name type and name induction. Such an implementation would make nominal-style mechanized metatheory accessible to practitioners. Reasoning in parametric nominal type theory often requires computing the translation on terms and the theory of Chapter 3 is interval-based, meaning that SAP witnesses are built by hand. An observational variant (or to a lesser degree a ROTT-like solution) could solve this scalability issue.

Automation of SRP and SAP obligations Even with ROTT, the user must express the target type explicitly as a relativistic reflexive graph (RRG). For types involving many nested type formers this can be verbose. A tactic or elaboration mechanism that infers RRG witnesses automatically would improve the user experience and make the approach competitive with computational parametricity translations.

Short Curriculum Vitae

Antoine Van Muylder holds a bachelor's degree in mathematics from Université Libre de Bruxelles and a master's degree in mathematical logic and theoretical computer science from Université de Paris. During and after his master's, he worked as a research engineer at Inria Paris, contributing to tools for formal verification of software security. He then joined KU Leuven as a doctoral researcher under the supervision of Andreas Nuyts and Dominique Devriese. His doctoral thesis concerns parametricity, a pervasive mathematical principle helpful to prove the correctness of programs. He first contributed an implementation of internal parametricity for the Agda proof assistant, accompanied by a theoretical foundation. He subsequently explored a novel connection between parametricity and nominal type theory, leading to a framework for principled reasoning about names and binding in programming languages.

List of Publications

From most to least recent, excluding abstracts. The first entry has been accepted at the TYPES 2025 conference post-proceedings.

- A. Van Muylder et al. “Nominal Type Theory by Nullary Internal Parametricity”. In: *arXiv preprint arXiv:2512.09464* (2025). Accepted at the TYPES 2025 Post-Proceedings.
- A. Van Muylder et al. “Internal and Observational Parametricity for Cubical Agda”. In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 209–240. DOI: 10.1145/3632850. URL: https://doi.org/10.1145/3632850
- P. G. Haselwarter et al. “SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq”. In: *ACM Trans. Program. Lang. Syst.* 45.3 (2023), 15:1–15:61. DOI: 10.1145/3594735. URL: https://doi.org/10.1145/3594735
- C. Abate et al. “SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq”. In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00048. URL: https://doi.org/10.1109/CSF51468.2021.00048
- K. Maillard et al. “The next 700 relational program logics”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 4:1–4:33. DOI: 10.1145/3371072. URL: https://doi.org/10.1145/3371072

Bibliography

[AAG05] M. G. Abbott, T. Altenkirch, and N. Ghani. "Containers: Constructing strictly positive types". In: *Theor. Comput. Sci.* 342.1 (2005), pp. 3–27. DOI: 10.1016/J.TCS.2005.06.002. URL: https://doi.org/10.1016/j.tcs.2005.06.002.

[Aba+21] C. Abate, P. G. Haselwarter, E. Rivas, A. Van Muylder, T. Winterhalter, C. Hritcu, K. Maillard, and B. Spitters. "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq". In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00048. URL: https://doi.org/10.1109/CSF51468.2021.00048.

[Abe24] C. B. Aberlé. "Parametricity via Cohesion". In: *CoRR* abs/2404.03825 (2024). DOI: 10.48550/ARXIV.2404.03825. arXiv: 2404.03825. URL: https://doi.org/10.48550/arXiv.2404.03825.

[ACC93] M. Abadi, L. Cardelli, and P. Curien. "Formal Parametric Polymorphism". In: *Theor. Comput. Sci.* 121.1&2 (1993), pp. 9–58. DOI: 10.1016/0304-3975(93)90082-5. URL: https://doi.org/10.1016/0304-3975(93)90082-5.

[AFH18] C. Angiuli, K. H. (Favonia), and R. Harper. "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*. Ed. by D. R. Ghica and A. Jung. Vol. 119. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 6:1–6:17. DOI: 10.4230/LIPICS.CSL.2018.6. URL: https://doi.org/10.4230/LIPIcs.CSL.2018.6.

[AFS18] S. Awodey, J. Frey, and S. Speight. "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*. Ed. by A. Dawar and E. Grädel. ACM, 2018, pp. 76–85. DOI: 10.1145/3209108.3209130. URL: https://doi.org/10.1145/3209108.3209130.

[Agd] T. Agda Community. *A standard library for Cubical Agda.* URL: https://github.com/agda/cubical.

[Agd23] Agda Development Team. *Agda 2.6.3 documentation.* 2023. URL: https://agda.readthedocs.io/en/v2.6.3/.

[Agd25] Agda Development Team. *Agda 2.8.0 documentation.* 2025. URL: https://agda.readthedocs.io/en/v2.8.0/.

[AGJ14] R. Atkey, N. Ghani, and P. Johann. "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.* Ed. by S. Jagannathan and P. Sewell. ACM, 2014, pp. 503–516. DOI: 10.1145/2535838.2535852. URL: https://doi.org/10.1145/2535838.2535852.

[Ahr+20] B. Ahrens, P. R. North, M. Shulman, and D. Tsementzis. "A Higher Structure Identity Principle". In: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020.* Ed. by H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller. ACM, 2020, pp. 53–66. DOI: 10.1145/3373718.3394755. URL: https://doi.org/10.1145/3373718.3394755.

[AK15] T. Altenkirch and A. Kaposi. "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.* Ed. by T. Uustalu. Vol. 69. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, 3:1–3:27. DOI: 10.4230/LIPICS.TYPES.2015.3. URL: https://doi.org/10.4230/LIPIcs.TYPES.2015.3.

[AKS22] T. Altenkirch, A. Kaposi, and M. Shulman. "Towards Higher Observational Type Theory". In: *28th International Conference on Types for Proofs and Programs (TYPES 2022). University of Nantes.* 2022.

[AL19] B. Ahrens and P. L. Lumsdaine. "Displayed Categories". In: *Log. Methods Comput. Sci.* 15.1 (2019). DOI: 10.23638/LMCS-15(1:20)2019. URL: https://doi.org/10.23638/LMCS-15(1:20)2019.

[Alt+24] T. Altenkirch, Y. Chamoun, A. Kaposi, and M. Shulman. "Internal Parametricity, without an Interval". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 2340–2369. DOI: 10.1145/3632920. URL: https://doi.org/10.1145/3632920.

[ALY09] R. Atkey, S. Lindley, and J. Yallop. "Unembedding domain-specific languages". In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009.* Ed. by S. Weirich. ACM, 2009, pp. 37–48. DOI: 10.1145/1596638.1596644. URL: https://doi.org/10.1145/1596638.1596644.

[AM17] A. Anand and G. Morrisett. "Revisiting Parametricity: Inductives and Uniformity of Propositions". In: *CoRR* abs/1705.01163 (2017). arXiv: 1705.01163. URL: http://arxiv.org/abs/1705.01163.

[AMS07] T. Altenkirch, C. McBride, and W. Swierstra. "Observational equality, now!" In: *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*. Ed. by A. Stump and H. Xi. ACM, 2007, pp. 57–68. DOI: 10.1145/1292597.1292608. URL: https://doi.org/10.1145/1292597.1292608.

[Ang+21a] C. Angiuli, G. Brunerie, T. Coquand, R. Harper, K. H. (Favonia), and D. R. Licata. "Syntax and models of Cartesian cubical type theory". In: *Math. Struct. Comput. Sci.* 31.4 (2021), pp. 424–468. DOI: 10.1017/S0960129521000347. URL: https://doi.org/10.1017/S0960129521000347.

[Ang+21b] C. Angiuli, E. Cavallo, A. Mörtberg, and M. Zeuner. "Internalizing representation independence with univalence". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434293. URL: https://doi.org/10.1145/3434293.

[AP90] M. Abadi and G. D. Plotkin. "A Per Model of Polymorphism and Recursive Types". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pp. 355–365. DOI: 10.1109/LICS.1990.113761. URL: https://doi.org/10.1109/LICS.1990.113761.

[Atk09] R. Atkey. "Syntax for Free: Representing Syntax with Binding Using Parametricity". In: *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*. Ed. by P. Curien. Vol. 5608. Lecture Notes in Computer Science. Springer, 2009, pp. 35–49. DOI: 10.1007/978-3-642-02273-9_5. URL: https://doi.org/10.1007/978-3-642-02273-9%5C_5.

[Atk12] R. Atkey. "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*. Ed. by P. Cégielski and A. Durand. Vol. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 46–61. DOI: 10.4230/LIPICS.CSL.2012.46. URL: https://doi.org/10.4230/LIPIcs.CSL.2012.46.

[BAC95] R. Bellucci, M. Abadi, and P. Curien. "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*. Ed. by M. Dezani-Ciancaglini and G. D. Plotkin. Vol. 902. Lecture Notes in Computer Science. Springer, 1995, pp. 32–46. DOI: 10.1007/BFB0014043. URL: https://doi.org/10.1007/BFb0014043.

[BB85] C. Böhm and A. Berarducci. "Automatic synthesis of typed λ-programs on term algebras". In: *Theoretical Computer Science* 39 (1985), pp. 135–154.

[BCM15] J. Bernardy, T. Coquand, and G. Moulin. "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*. Ed. by D. R. Ghica. Vol. 319. Electronic Notes in Theoretical Computer Science. Elsevier, 2015, pp. 67–82. DOI: 10.1016/J.ENTCS.2015.12.006. URL: https://doi.org/10.1016/j.entcs.2015.12.006.

[BHL20] A. Bauer, P. G. Haselwarter, and P. L. Lumsdaine. "A general definition of dependent type theories". In: *CoRR* abs/2009.05539 (2020). arXiv: 2009.05539. URL: https://arxiv.org/abs/2009.05539.

[BJP12] J. Bernardy, P. Jansson, and R. Paterson. "Proofs for free - Parametricity for dependent types". In: *J. Funct. Program.* 22.2 (2012), pp. 107–152. DOI: 10.1017/S0956796812000056. URL: https://doi.org/10.1017/S0956796812000056.

[BKS23] R. Bocquet, A. Kaposi, and C. Sattler. "For the Metatheory of Type Theory, Internal Sconing Is Enough". In: *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, Rome, Italy, July 3-6, 2023*. Ed. by M. Gaboardi and F. van Raamsdonk. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 18:1–18:23. DOI: 10.4230/LIPICS.FSCD.2023.18. URL: https://doi.org/10.4230/LIPIcs.FSCD.2023.18.

[BM12] J. Bernardy and G. Moulin. "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, 2012, pp. 135–144. DOI: 10.1109/LICS.2012.25. URL: https://doi.org/10.1109/LICS.2012.25.

[BM13] J. Bernardy and G. Moulin. "Type-theory in color". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by G. Morrisett and T. Uustalu. ACM, 2013, pp. 61–72. DOI: 10.1145/2500365.2500577. URL: https://doi.org/10.1145/2500365.2500577.

[Boo+16] A. B. Booij, M. H. Escardó, P. L. Lumsdaine, and M. Shulman. "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*. Ed. by S. Ghilezan, H. Geuvers, and J. Ivetic. Vol. 97. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 7:1–7:14. DOI: 10.4230/LIPICS.TYPES.2016.7. URL: https://doi.org/10.4230/LIPIcs.TYPES.2016.7.

[BP09] J. Bengtson and J. Parrow. "Formalising the pi-calculus using nominal logic". In: *Log. Methods Comput. Sci.* 5.2 (2009). URL: http://arxiv.org/abs/0809.3960.

[BPT17] S. Boulier, P. Pédrot, and N. Tabareau. "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*. Ed. by Y. Bertot and V. Vafeiadis. ACM, 2017, pp. 182–194. DOI: 10.1145/3018610.3018620. URL: https://doi.org/10.1145/3018610.3018620.

[Cav20] E. Cavallo. *Ptt, an experimental implementation of Martin-Löf type theory with n-ary internal parametricity*. 2020. URL: https://github.com/ecavallo/ptt.

[CH19] E. Cavallo and R. Harper. "Parametric Cubical Type Theory". In: *CoRR* abs/1901.00489 (2019). arXiv: 1901.00489. URL: http://arxiv.org/abs/1901.00489.

[CH21] E. Cavallo and R. Harper. "Internal Parametricity for Cubical Type Theory". In: *Log. Methods Comput. Sci.* 17.4 (2021). DOI: 10.46298/LMCS-17(4:5)2021. URL: https://doi.org/10.46298/lmcs-17(4:5)2021.

[Che12] J. Cheney. "A dependent nominal type theory". In: *Log. Methods Comput. Sci.* 8.1 (2012). DOI: 10.2168/LMCS-8(1:8)2012. URL: https://doi.org/10.2168/LMCS-8(1:8)2012.

[Chl08] A. Chlipala. "Parametric higher-order abstract syntax for mechanized semantics". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by J. Hook and P. Thiemann. ACM, 2008, pp. 143–156. DOI: 10.1145/1411204.1411226. URL: https://doi.org/10.1145/1411204.1411226.

[Coh+17] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *FLAP* 4.10 (2017), pp. 3127–3170. URL: http://collegepublications.co.uk/ifcolog/?00019.

[Coq22] Coq development team. *The Coq proof assistant*. 2022. URL: http://coq.inria.fr.

[Gir72] J.-Y. Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Éditeur inconnu, 1972.

[Gir86] J. Girard. "The System F of Variable Types, Fifteen Years Later". In: *Theor. Comput. Sci.* 45.2 (1986), pp. 159–192. DOI: 10.1016/0304-3975(86)90044-7. URL: https://doi.org/10.1016/0304-3975(86)90044-7.

[Has+23] P. G. Haselwarter, E. Rivas, A. Van Muylder, T. Winterhalter, C. Abate, N. Sidorenco, C. Hritcu, K. Maillard, and B. Spitters. "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq". In: *ACM Trans. Program. Lang. Syst.* 45.3 (2023), 15:1–15:61. DOI: 10.1145/3594735. URL: https://doi.org/10.1145/3594735.

[Hof97] M. Hofmann. "Syntax and semantics of dependent types". In: *Extensional Constructs in Intensional Type Theory*. London: Springer London, 1997, pp. 13–54. DOI: 10.1007/978-1-4471-0963-1_2.

[Hof99] M. Hofmann. "Semantical Analysis of Higher-Order Abstract Syntax". In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 1999, pp. 204–213. DOI: 10.1109/LICS.1999.782616. URL: https://doi.org/10.1109/LICS.1999.782616.

[HRR13] C. Hermida, U. S. Reddy, and E. P. Robinson. "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*. Ed. by J. Power and C. Wingfield. Vol. 303. Electronic Notes in Theoretical Computer Science. Elsevier, 2013, pp. 149–180. DOI: 10.1016/J.ENTCS.2014.02.008. URL: https://doi.org/10.1016/j.entcs.2014.02.008.

[HS97] M. Hofmann and T. Streicher. *Lifting Grothendieck Universes*. Unpublished note. 1997.

[Kap25] A. Kaposi. "Second-order generalised algebraic theories, by examples (under construction)". In: (2025).

[KD13] N. R. Krishnaswami and D. Dreyer. "Internalizing Relational Parametricity in the Extensional Calculus of Constructions". In: *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*. Ed. by S. R. D. Rocca. Vol. 23. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 432–451. DOI: 10.4230/LIPICS.CSL.2013.432. URL: https://doi.org/10.4230/LIPIcs.CSL.2013.432.

[KL12] C. Keller and M. Lasson. "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*. Ed. by P. Cégielski and A. Durand. Vol. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 381–395. DOI: 10.4230/LIPICS.CSL.2012.381. URL: https://doi.org/10.4230/LIPIcs.CSL.2012.381.

[Lei91] D. Leivant. "Finitely Stratified Polymorphism". In: *Inf. Comput.* 93.1 (1991), pp. 93–113. DOI: 10.1016/0890-5401(91)90053-5. URL: https://doi.org/10.1016/0890-5401(91)90053-5.

[Mai+20] K. Maillard, C. Hritcu, E. Rivas, and A. Van Muylder. "The next 700 relational program logics". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 4:1–4:33. DOI: 10.1145/3371072. URL: https://doi.org/10.1145/3371072.

[Mak95] M. Makkai. "First order logic with dependent sorts, with applications to category theory". In: (1995). URL: http://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf.

[Mar+10] S. Marlow et al. "Haskell 2010 language report". In: *Available online http://www. haskell. org/(May 2011)* (2010).

[MM18] B. Mannaa and R. E. Møgelberg. "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*. Ed. by H. Kirchner. Vol. 108. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 23:1–23:17. DOI: 10.4230/LIPICS.FSCD.2018.23. URL: https://doi.org/10.4230/LIPIcs.FSCD.2018.23.

[Mou16] G. Moulin. "Internalizing Parametricity". PhD thesis. Chalmers University of Technology, Gothenburg, Sweden, 2016. URL: http://publications.lib.chalmers.se/publication/235758-internalizing-parametricity.

[MP88] J. C. Mitchell and G. D. Plotkin. "Abstract Types Have Existential Type". In: *ACM Trans. Program. Lang. Syst.* 10.3 (1988), pp. 470–502. DOI: 10.1145/44501.45065. URL: https://doi.org/10.1145/44501.45065.

[MPW92] R. Milner, J. Parrow, and D. Walker. "A Calculus of Mobile Processes, I". In: *Inf. Comput.* 100.1 (1992), pp. 1–40. DOI: 10.1016/0890-5401(92)90008-4. URL: https://doi.org/10.1016/0890-5401(92)90008-4.

[MU21] L. d. Moura and S. Ullrich. "The lean 4 theorem prover and programming language". In: *International Conference on Automated Deduction*. Springer. 2021, pp. 625–635.

[Nar25a] Narya development team. *Narya Documentation*. Accessed: 2025-11-19. Nov. 2025. URL: https://narya.readthedocs.io/en/latest/.

[Nar25b] Narya development team. *Nominal Syntax — Narya Documentation*. Accessed: 2025-11-19. Nov. 2025. URL: https://narya.readthedocs.io/en/latest/nominal.html.

[ND18] A. Nuyts and D. Devriese. "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*. Ed. by A. Dawar and E. Grädel. ACM, 2018, pp. 779–788. DOI: 10.1145/3209108.3209119. URL: https://doi.org/10.1145/3209108.3209119.

[ND24] A. Nuyts and D. Devriese. "Transpension: The Right Adjoint to the Pi-type". In: *Log. Methods Comput. Sci.* 20.2 (2024). DOI: 10.46298/LMCS-20(2:16)2024. URL: https://doi.org/10.46298/lmcs-20(2:16)2024.

[Nor07] U. Norell. "Towards a Practical Programming Language Based on Dependent Type Theory". PhD thesis. Chalmers, 2007.

[Nuy21] A. Nuyts. "Parametricity Features and their Requirements". In: *CoRR* abs/2111.09822 (2021). arXiv: 2111.09822. URL: https://arxiv.org/abs/2111.09822.

[Nuy25] A. Nuyts. "Transpension for Cubes without Diagonals". In: *Workshop on Homotopy Type Theory / Univalent Foundations*. 2025. URL: https://lirias.kuleuven.be/retrieve/803969.

[NVD17] A. Nuyts, A. Vezzosi, and D. Devriese. "Parametric quantifiers for dependent type theory". In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 32:1–32:29. DOI: 10.1145/3110276. URL: https://doi.org/10.1145/3110276.

[Ode94] M. Odersky. "A Functional Theory of Local Names". In: *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. Ed. by H. Boehm, B. Lang, and D. M. Yellin. ACM Press, 1994, pp. 48–59. DOI: 10.1145/174675.175187. URL: https://doi.org/10.1145/174675.175187.

[PA93] G. D. Plotkin and M. Abadi. "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*. Ed. by M. Bezem and J. F. Groote. Vol. 664. Lecture Notes in Computer Science. Springer, 1993, pp. 361–375. DOI: 10.1007/BFB0037118. URL: https://doi.org/10.1007/BFb0037118.

[Pit00] A. M. Pitts. "Parametric polymorphism and operational equivalence". In: *Math. Struct. Comput. Sci.* 10.3 (2000), pp. 321–359. URL: http://journals.cambridge.org/action/displayAbstract?aid=44651.

[Pit03] A. M. Pitts. "Nominal logic, a first order theory of names and binding". In: *Inf. Comput.* 186.2 (2003), pp. 165–193. DOI: 10.1016/S0890-5401(03)00138-X. URL: https://doi.org/10.1016/S0890-5401(03)00138-X.

[Pit10] A. M. Pitts. "Nominal system T". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by M. V. Hermenegildo and J. Palsberg. ACM, 2010, pp. 159–170. DOI: 10.1145/1706299.1706321. URL: https://doi.org/10.1145/1706299.1706321.

[Pit13] A. M. Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

[Pit14] A. Pitts. “Nominal Sets and Dependent Type Theory”. In: *TYPES*. 2014. URL: https://www.irif.fr/~letouzey/types2014/slides-inv3.pdf.

[PMD14] A. M. Pitts, J. Matthiesen, and J. Derikx. “A Dependent Type Theory with Abstractable Names”. In: *Ninth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2014, Brası´lia, Brazil, September 8-9, 2014*. Ed. by M. Ayala-Rincón and I. Mackie. Vol. 312. Electronic Notes in Theoretical Computer Science. Elsevier, 2014, pp. 19–50. DOI: 10.1016/J.ENTCS.2015.04.003. URL: https://doi.org/10.1016/j.entcs.2015.04.003.

[PT22] L. Pujet and N. Tabareau. “Observational equality: now for good”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–27. DOI: 10.1145/3498693. URL: https://doi.org/10.1145/3498693.

[Reb25] S. Reboullet. “Universe discrepancies in nominal presheaf models of parametric type theory”. Theses. Université Paris Cité, Sept. 2025. URL: https://theses.hal.science/tel-05584055.

[Rey74] J. C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. Ed. by B. J. Robinet. Vol. 19. Lecture Notes in Computer Science. Springer, 1974, pp. 408–423. DOI: 10.1007/3-540-06859-7_148. URL: https://doi.org/10.1007/3-540-06859-7%5C_148.

[Rey83] J. C. Reynolds. “Types, Abstraction and Parametric Polymorphism”. In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A. Mason. North-Holland/IFIP, 1983, pp. 513–523.

[Rey84] J. C. Reynolds. “Polymorphism is not Set-Theoretic”. In: *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*. Ed. by G. Kahn, D. B. MacQueen, and G. D. Plotkin. Vol. 173. Lecture Notes in Computer Science. Springer, 1984, pp. 145–156. DOI: 10.1007/3-540-13346-1_7. URL: https://doi.org/10.1007/3-540-13346-1%5C_7.

[Rij22] E. Rijke. *Introduction to Homotopy Type Theory*. 2022. arXiv: 2212.11082 [math.LO].

[Roc25] T. R. D. Team. *The Rocq Prover*. Version 9.1.0. Sept. 2025. DOI: 10.5281/zenodo.17473943. URL: https://doi.org/10.5281/zenodo.17473943.

[RR94] E. P. Robinson and G. Rosolini. “Reflexive Graphs and Parametric Polymorphism”. In: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS ’94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 364–371. DOI: 10.1109/LICS.1994.316053. URL: https://doi.org/10.1109/LICS.1994.316053.

[RWL22] R. Rose, M. Z. Weaver, and D. R. Licata. "Deciding the cofibration logic of cartesian cubical type theories". In: *28th International Conference on Types for Proofs and Programs (TYPES 2022). University of Nantes*. 2022.

[Sch+24] R. Schepens, D. Devriese, A. Nuyts, A. Van Muylder, and K. L. F. W. O. M. of Mathematics (Leuven) degree granting institution. *Effect Parametricity and Other Free Theorems in a Parametric Proof Assistant*. eng. Leuven, 2024.

[Sch06] U. Schöpp. "Names and binding in type theory". PhD thesis. University of Edinburgh, UK, 2006. URL: https://hdl.handle.net/1842/1203.

[SPG03] M. R. Shinwell, A. M. Pitts, and M. Gabbay. "FreshML: programming with binders made simple". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. Ed. by C. Runciman and O. Shivers. ACM, 2003, pp. 263–274. DOI: 10.1145/944705.944729. URL: https://doi.org/10.1145/944705.944729.

[SS04] U. Schöpp and I. Stark. "A Dependent Type Theory with Names and Binding". In: *Computer Science Logic*. Ed. by J. Marcinkowski and A. Tarlecki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 235–249. ISBN: 978-3-540-30124-0. DOI: 10.1007/978-3-540-30124-0_20.

[Tak01] I. Takeuti. *The Theory of Parametricity in Lambda Cube*. Technical report 1217, Kyoto University. 2001. URL: https://repository.kulib.kyoto-u.ac.jp/dspace/bitstream/2433/41237/1/1217_10.pdf.

[TTS21] N. Tabareau, É. Tanter, and M. Sozeau. "The Marriage of Univalence and Parametricity". In: *J. ACM* 68.1 (Jan. 2021). ISSN: 0004-5411. DOI: 10.1145/3429979. URL: https://doi.org/10.1145/3429979.

[Uni13] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://homotopytypetheory.org/book/.

[Urb08] C. Urban. "Nominal Techniques in Isabelle/HOL". In: *J. Autom. Reason.* 40.4 (2008), pp. 327–356. DOI: 10.1007/S10817-008-9097-2. URL: https://doi.org/10.1007/s10817-008-9097-2.

[VMA21] A. Vezzosi, A. Mörtberg, and A. Abel. "Cubical Agda: A dependently typed programming language with univalence and higher inductive types". In: *J. Funct. Program.* 31 (2021), e8. DOI: 10.1017/S0956796821000034. URL: https://doi.org/10.1017/S0956796821000034.

[VND23] A. Van Muylder, A. Nuyts, and D. Devriese. *Agda --bridges VM*. 2023. DOI: https://doi.org/10.5281/zenodo.8413157. URL: https://zenodo.org/records/8413157.

[VND24] A. Van Muylder, A. Nuyts, and D. Devriese. "Internal and Observational Parametricity for Cubical Agda". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 209–240. DOI: 10.1145/3632850. URL: https://doi.org/10.1145/3632850.

[VND25] A. Van Muylder, A. Nuyts, and D. Devriese. "Nominal Type Theory by Nullary Internal Parametricity". In: *arXiv preprint arXiv:2512.09464* (2025). Accepted at the TYPES 2025 Post-Proceedings.

[Voi09] J. Voigtländer. "Free theorems involving type constructor classes: functional pearl". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by G. Hutton and A. P. Tolmach. ACM, 2009, pp. 173–184. DOI: 10.1145/1596550.1596577. URL: https://doi.org/10.1145/1596550.1596577.

[VV20] N. Veltri and A. Vezzosi. "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*. Ed. by J. Blanchette and C. Hritcu. ACM, 2020, pp. 270–283. DOI: 10.1145/3372885.3373814. URL: https://doi.org/10.1145/3372885.3373814.

[VV23] N. Veltri and A. Vezzosi. "Formalizing CCS and π-calculus in Guarded Cubical Agda". In: *J. Log. Algebraic Methods Program.* 131 (2023), p. 100846. DOI: 10.1016/J.JLAMP.2022.100846. URL: https://doi.org/10.1016/j.jlamp.2022.100846.

[Wad07] P. Wadler. "The Girard-Reynolds isomorphism (second edition)". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 201–226. DOI: 10.1016/J.TCS.2006.12.042. URL: https://doi.org/10.1016/j.tcs.2006.12.042.

[Wad89] P. Wadler. "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*. Ed. by J. E. Stoy. ACM, 1989, pp. 347–359. DOI: 10.1145/99370.99404. URL: https://doi.org/10.1145/99370.99404.

Statement on the use of Generative AI

I did not use generative AI assistance tools during the research/writing process of my thesis, except for mere language assistance. The use of GenAI tools for language assistance does not need to be further specified.
The text, code, and images in this thesis are my own (unless otherwise specified). Generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
Celestijnenlaan 200A box 2402
B-3001 Leuven
http://www.distrinet.cs.kuleuven.be