Church-encodings in higher-dimensional type theory
Toward a theory of abstraction

C.B. Aberlé

The rising sea

Alexander Grothendieck famously described his approach to mathematical problem-solving with the following metaphor:

> A different image came to me a few weeks ago. The unknown thing to be known appeared to me as some stretch of earth or hard marl, resisting penetration… the sea advances insensibly in silence, nothing seems to happen, nothing moves, the water is so far off you hardly hear it… yet it finally surrounds the resistant substance.[1]

Among Grothendieck's singular talents, widely admired was (and still is) his knack for *abstraction* – his ability to find *just* the right level of generality from which to attack a problem, to 'paint a landscape in which the proof is obvious', as Pierre Deligne once put it.[2]

The past four years of my intellectual life have been dominated by the urge to better understand just what such *abstraction* consists in. Initially, the problem posed itself to me in my first forays into computer science – any half-decent introduction to programming will make at least a passing mention of abstraction and the essential role it plays in managing the complexity of programs. Yet for all its ubiquity, such talk of abstraction tends to be light on detail. Even those texts that make abstraction a central topic, such as Abelson & Sussman's classic *The Structure and Interpretation of Computer Programs* (SICP)[1], never explicitly define abstraction, but rather illustrate the concept and its utility by way of multifarious examples. The way one is meant to become acquainted with this notion is evidently by *ostension*.

The examples I encountered in SICP and elsewhere impressed upon me the *significance* and *power* of abstraction as a conceptual tool, yet left me unsatisfied – for though I came to be acquainted with many *instances* of abstraction, I could not, in general, say what it was for one thing to be an abstraction of another, nor could I say, given a particular problem to be solved, what sort of abstraction would be *the right tool for the job*, as it were. I became convinced of the need for a *theory of abstraction* capable of answering these questions, if only to quiet my own curiosities. Such a theory, as I see it, would provide precise answers to both of the following, related questions:

1. "*What* is abstraction?", an answer to which would be a *mathematical* characterisation of abstraction, *general* enough to cover most (or at least a significant portion) of the examples to which the word 'abstraction' is commonly applied in e.g. computer science, yet *robust* enough to yield novel insights, to suggest new kinds of abstraction, etc.

[1] this quote is extracted from in Grothendieck's manuscript *Rècoltes et Semailles.*

[2] Pierre Deligne, *Theorie des topos et cohomologie etale des schemas, Tome 3, p. 584*

2. "*How* are abstractions of the sort identified above useful in solving problems?", an answer to which, in turn, would suggest criteria for assessing what sort of abstraction would be best-suited for solving a given problem, or class of problems.

My attempts to answer these questions ultimately led me to the object of this report: developing a type theory capable of proving certain *parametricity* theorems.

Types, abstraction & parametric polymorphism

From the outset of my study of abstraction, I had some sense that the notions of abstraction in computer science, mathematics, and logic were closely connected – if not identical – and that a *theory of abstraction*, properly developed, should apply equally to all three domains. Grothendieck's metaphor of the rising sea is reminiscent of Abelson & Sussman's concept of *metalinguistic abstraction* – solving a problem by establishing a *language* in which both the problem and its solution can naturally be expressed. However, it was only once I learned of the correspondence between proofs/propositions in (constructive) logic & mathematics, and programs/types in computer science, respectively (the so-called Curry-Howard correspondence) that I began to see how these various notions of abstraction might be unified. This view was crystallised for me by the following remark made by John Reynolds in his seminal 1983 paper "Types, Abstraction and Parametric Polymorphism":[2]

Type structure is a syntactic discipline for enforcing levels of abstraction.

Reynolds' remark is only informal and suggestive, but a particularly fruitful interpretation takes 'type structure' to mean the structure of computable functions that exist between types in a programming language, which, by the Curry-Howard correspondence, is just the same thing as the structure of proofs in the corresponding logic. Such a structure is a paradigmatic example of a *category*, an abstract mathematical structure, and the namesake of *category theory*. One of my earlier realisations in the course of undertaking this project was that *any* category could in fact be viewed as similarly defining a structure of *levels of abstraction*, in the Reynoldsian sense. What remains to be seen, then, is what additional structure a category must possess in order to enable the particular *kinds* of abstraction employed in computer science, mathematics, etc., and how this relates to our ordinary, informal ideas about abstraction, e.g. that abstraction suppresses irrelevant information, reduces complexity, etc. This was much of the motivation for Reynolds' 1983 paper – to better understand the properties of certain type-systems that make them particularly well-behaved with respect to abstraction.

In particular, within Reynolds' object of study, the polymorphic λ-calculus (which corresponds to Girard's System F of second-order intuitionistic logic), it is possible, for any property of types expressible within the system, to form a type which is *optimal* (in the sense of being a category-theoretic *limit/colimit*) with respect to the structure of types with that property. Types of this form have a computational interpretation as the types of *Church-encodings* of abstract data types in λ-calculus, which can be used to represent all manner of mathematical or computational structures, e.g. natural numbers, tuples, lists, trees, etc. This relates type structure to another informal idea about abstraction: that abstraction allows us to form *ideal representations* of various patterns and properties.

The study of Church-encodings therefore seems a promising avenue toward a theory of abstraction. Proving that such encodings have the desired mathematical properties of the structures they are meant to model can be highly non-trivial, however. Toward this end, Reynolds introduced the concept of *relational parametricity*, which has proved an invaluable tool in the analysis of type systems. Reynolds' technique was adapted by Philip Wadler to prove a wealth of 'theorems for free'[3] about programs based solely on their types. Significantly, Wadler was able to show that the two principles of *function extensionality* (i.e. two functions are equal if their outputs are equal for all possible inputs) and *parametricity* jointly imply an induction principle for the System F encoding of the natural numbers, and hence that this encoding satisfies all the axioms of arithmetic[4]. These *free theorems* thus offer a promising candidate for the kind of structure we might take to be characteristic of type-theoretic abstraction.

Such theorems, however, remain stubbornly meta-theoretic. One cannot even *express* a principle of induction for natural numbers within System F, since System F lacks types that can depend on values (so-called *dependent types*), and the most straightforward way of extending System F with dependent types – Coquand and Huet's Calculus of Constructions – while capable of expressing such a principle, is incapable of proving it. For this reason in particular, modern type theories have generally eschewed Church encodings of abstract data types in favour of the more restricted alternative of inductive datatypes, which explicitly include induction principles as axioms of the theory, but are restricted to a narrower class of mathematical structures. Church encodings hold the potential to account for a much broader class of mathematical/computational abstractions, so it is worth investigating the possibility of developing a type theory that internalizes the kinds of reasoning embodied in Reynolds' and Wadler's proofs (namely: function extensionality and relational parametricity). Such a type theory could, as I see it, form a central component of a more general theory of abstraction, in that it would provide:

1. A theoretical framework for exploring the properties of various computational and mathematical abstractions by modelling these abstractions as Church encodings and proving that they obey certain *free theorems.* This, in turn, may suggest novel forms of abstraction that may form the objects of further study.
2. By considering the logical and computational devices involved in proving such *free theorems*, a more *general* characterisation of what structure a category must possess in order for similar *free theorems* to hold of its objects. What I have in mind is roughly analogous to how the concept of a *topos* generalises the category of sets as 'an abstract context in which one can do mathematics'[3]. This general characterisation would likely be obtained by considering (suitable generalisations of) the class of models of the type theory.
3. By way of proof-theoretic analysis of the type theory, a means of studying *how* abstractions are of use in logic, computation, and mathematics. E.g. once the type theory is established, we can sensibly ask whether a proof/program that makes use of some abstraction is *shorter* than one that does not use that same abstraction, and hence whether or not that abstraction *reduces the complexity* of the proof/program.

Internalising either of function extensionality or relational parametricity, let alone both, presents a significant challenge, but recent developments in type theory offer promising points of attack for these problems. The type theory I have developed can be seen as a synthesis of two parallel currents in type theory:

- The development of various type theories, which, following Atkey, I call *quantitative type systems*, that extend the resource-sensitive behaviour of Girard's Linear Logic to allow fine control over the use of resources within the theory.

[3]this charactersiation of toposes appears on the nLab page for toposes: https://ncatlab.org/nlab/show/topos

- Voevodsky et al.'s programme of Homotopy Type Theory, and the subsequent development of Cubical Type Theory.

In developing this theory, my goal has been to ensure that the following **key properties** of the theory all hold:

- *Normalisation*: every computation terminates and yields a term in *normal form*.
- *Confluence*: if there are multiple distinct terms that a given term computes to, then further computation can be done to reduce these to a common term.
- *Subject reduction*: validity of proofs/programs is preserved by computation of their constituent terms.
- *Canonicity*: every term of a given type in normal form is explicitly built up from the constructors of that type.
- *Decidability* of proof/type checking: there is an algorithm, that always terminates, for determining whether a given proof/program is valid.

Taken together, these properties ensure the coherence of the theory both as a computational system and as a logic. What follows is an outline of my process for developing this theory, and of my attempts at proving that the above properties hold of it.

A Theory of Types

Having initially sketched a plan for the design of the theory, and the various logical/computational devices I expected to use to accomplish my goals, I proceeded to build the theory up in stages, starting from a basic dependent type theory in the style of Per Martin-Löf, and working up to the full theory with all the whistles and bells. This has allowed me to adopt the following strategy for proving that the theory has all desired properties – in the first stage, I prove that these properties hold of the basic theory; in subsequent stages, whenever the theory is extended by some new capability, I show how to adapt the prior proofs of these properties to the modified theory. At present, most of my proofs are only sketches. Further work will be required to flesh these out into full proofs, or better yet, to formalise the proofs in a proof assistant.

First stage – the basic type theory

In the first stage, my research into the design of systems of dependent type theory led me to adopt a *bidirectional* typing discipline. The core idea of bidirectional type-checking is to treat the judgments of type theory not merely as statements, but as implicitly-defined *programs* that take some input and yield some output (which need not be merely true or false). A type theory in bidirectional style thus has distinct judgments for type-*checking* (in which both the term to be checked and the proposed type are given as inputs) and type-*synthesis* (in which only the term being checked is provided as input, and – if the term is well-formed – a type for that term is yielded as output); the interplay between these two judgments forms the motor that drives the whole operation of bidirectional typing. Hence in a bidirectional system the usual typing judgment $t : T$ (t is of type T) is split in twain:

- The judgment $T \ni t$ means that term t can be *checked* to be of type T.
- The judgment $e \in R$ means that type R can be *synthesised* for term e.

The practical upshot of distinguishing these judgments from one another comes into play when laying out the rules for validity of judgments: such rules implicitly define algorithms for type checking and synthesis, and this serves to show that type-checking for the system thereby defined is at least semi-decidable. More broadly, the bidirectional discipline has a tendency of making metatheoretic proofs relatively straightforward and painless, where they might otherwise be quite involved under more traditional styles of presentation. An additional virtue of bidirectional style, at least for the basic theory, is that it keeps the overhead of typing information a programmer must supply to the typechecker to a minimum.

My main references for learning the principles of bidirectional type systems have been Conor McBride's various writings, talks, etc. on the topic,[5],[10] along with additional writings by Neel Krishnaswami and Jana Dunfield.[6] Upon this foundation, I laid out a basic type theory in bidirectional style, with dependent function types (Π-types, corresponding to universal quantification), dependent pair types (Σ-types, corresponding to existential quantification), and a single universe à la Russell (i.e. a type whose elements are themselves types), where computation of terms corresponds to β-reduction.

I chose to make the basic theory predicative, as this enables a very clean proof of normalisation. However, since Church encodings – my ultimate object of study – rely on a certain amount of impredicativity, the final system must break the predicativity of the systems at previous stages and so also the validity of this proof. Nonetheless, so long as the proof otherwise remains valid right up until the theory becomes impredicative, this shows that the theory can only fail to be normalising as a result of this impredicativity. The manner of impredicativity permitted by the full theory is just the same as that allowed by System F and the Calculus of Constructions (i.e. impredicative quantification over types), which are themselves normalising, so we would similarly expect the full theory to be normalising. However, proving as much requires a significantly stronger mathematical arsenal than that involved in the proof of normalisation for the predicative theory, due to the massive increase in proof-theoretic strength resulting from impredicativity.

These considerations out of the way, the proofs of the **key properties** for the basic theory are all straightforward.

For the proof of normalisation, I adapted a method outlined by Chris Casinghino for proving normalisation of the Calculus of Constructions, as a simplification and unification of proofs by Geuvers, Nederhof, Melliès, and Werner.[7] These proofs all pertained to the Calculus of Constructions, an impredicative theory, but for my purposes, I was able to adapt the general method identified by Casinghino to extend to the dependently-typed setting a standard proof of normalisation for predicative theories *without* dependent types – assign a natural number to each type that measures its complexity, and then show that computation at a given type can only lead to computations at types of *strictly lower* complexity. The trick, as in the cases considered by Casinghino, is not only to define an interpretation of types into natural numbers, but also of terms, such that a function on types becomes a function on natural numbers, a pair of types becomes a pair of natural numbers, etc. Once this complexity measure has been defined, bidirectionalism takes care of the rest: a natural consequence of the bidirectional style is that every site of computation in a term is labelled with the type at which that computation is active. A simple case analysis on the rules for computation shows that the labels in a computed term (the contractum) are of strictly lower complexity than the label of the term it was computed from (the redex).

I have not encountered a proof of normalisation for a predicative dependent type theory elsewhere in the literature that uses this same method of directly assigning a complexity measure to types, so it may be of some theoretical interest. For my part, I have found that this method of proof scales nicely to some more advanced theories; in particular, for some of the type theories developed at later stages, it is not as yet clear how to extend interpretations of the sort considered by Casinghino, but it is entirely straightforward to extend the above method.

For the proof of confluence, I followed McBride's adaptation to the bidirectional setting of Takahashi's method of proving confluence (itself a simplification of the Tait/Martin-Löf method): define a notion of 'parallel reduction' that allows multiple redexes to be contracted at once, then show that this relation is confluent, and that its reflexive transitive closure coincides with that of ordinary computation, so ordinary computation is also confluent. The bidirectional discipline ensures that there is no overlap between redexes in a term, and there is exactly one way of reducing each redex. Hence a parallel computation corresponds uniquely to a choice of which redexes within a term to reduce. Any redex that was not reduced during one step of computation remains available for reduction at the next step. Hence any divergent parallel-computations can be unified by applying all reductions in the one that were not applied in the other, and vice versa.

Proving subject reduction for the basic theory is a simple matter of induction on derivations, along with some lemmas (e.g. that substitution of terms for free variables preserves typability) similarly proved by induction on derivations. As is often the case with proofs by induction, the only artful matter is the choice of induction hypothesis – everything else is elementary.

To prove canonicity and decidability, I construct a modified form of the type theory – I remove the rule that allows labelling a term with its type and the rules for performing computation in types, and instead ensure, using the proof of normalisation, that all terms and types are pre-computed to normal form. By subject reduction, if a term typechecks in the original theory, then its normal form typechecks in the modified theory. Simple inspection of the rules in this modified theory reveals that the only way to build up a term is using the constructors of its type (canonicity), and that typechecking a term involves only typechecking its strict subterms, hence the algorithm for typechecking must terminate (decidability).

Second stage – quantitative type systems

Having developed and proved the key properties of the basic theory, I next set out to lay the groundwork for internalising parametricity. Reynolds' concept of parametricity was based on the observation that type variables in the polymorphic λ-calculus cannot affect the *values* of computed terms, but may only be used to *parameterise* the types of such terms. The first step to internalising reasoning via parametricity is therefore to avail ourselves of a type system capable of distinguishing *parametric* use of variables from *value-relevant* use.

My two main references in developing such a theory have been Mishra-Linger,[8] and McBride.[9] Mishra-Linger directly extends the standard formalism of Pure Type Systems to include a notion of parametric quantification, and parametric functions. McBride, on the other hand, develops a more general framework of dependent type theories that are resource-sensitive in a sense closely connected to Girard's Linear Logic. Following Atkey,[10] I refer to a type system of the sort constructed by McBride as a quantitative type system (QTS).

The essential idea of a QTS is to annotate the usual typing judgments and variable bindings of a type theory with elements of some algebraic structure, interpreted as *resources* or *generalised quantities* that induce corresponding *quantitative modalities* on the use of terms they annotate. The interaction of such modalities with one another is then determined by the *algebraic structure* of their representative quantities. McBride's key insight was that such a structure of resources ought to include a resource representing *nothing*, so that the use of a variable in parameterising a type costs *nothing*. That is, if the algebraic structure of resources includes a suitable operation of *addition* that represents the simultaneous availability of resources, then the *unit* (i.e. zero) of this operation lets us mark those variables/terms that are *never* used in positions that affect the values of terms (i.e. are *value-irrelevant*) but that may still appear in the *types* of terms. Hence variables marked with the zero resource are *parametrically* quantified.

McBride proposed the theory of QTSs as a means of extending the resource-sensitive behaviour of Linear Logic and linear type systems to the dependently typed setting without sacrificing expressivity, but for my purposes, it offers a promising level of generality from which to view the problem of developing a type theory with parametric quantifiers, along the lines of Mishra-Linger. In particular, the theory of quantitative type systems clarifies many of the choices involved in the design of such a system that might otherwise seem arbitrary, and many desirable properties of the system fall out naturally as a result.

However, as noted by Atkey, McBride's original system suffers from a bug that renders some desirable meta-theoretic properties false. I have therefore found it necessary to refine and modify McBride's system somewhat. In fact, these modifications have the effect of simplifying the metatheory of QTSs, and unifying the framework with other attempts at similar goals, e.g. Mishra-Linger's thesis. These adjustments in place, the methods previously developed for proving the **key properties** go off without a hitch, as they are all nearly unchanged from the analogous proofs for the basic theory.

It is tempting, given a system with parametric quantification, to think that, since the *input* to a parametric function never gets used in a value-relevant way, the *outputs* of such a function must all be *equal* in some suitable sense. In fact, this idea is *key* to the method I adopt for internalising parametricity theorems within such a theory; however, care must be taken in making this notion precise. Mishra-Linger developed a notion of *erasure* – such that the *erasure* of a term is obtained by *erasing* all subterms that are only used parametrically – and proposed a modified conversion rule that allowed the identification of types *up to erasure*. This allowed the internalisation of certain metatheorems within the theory of Pure Type Systems with erasure, e.g. the uniqueness of identity proofs. However, as was later discovered, this modification has the unfortunate consequence of rendering type checking for the theory undecidable. Much of my efforts in subsequent stages can be seen as attempts to overcome this obstacle – to develop a theory where parametric quantification induces a kind of equality, but where type checking remains decidable.

Third stage – syntactic extensionality: η-expansion

Having developed the framework of QTSs in order to express parametric quantification, my next task was to equip the type theory I developed within this framework with a notion of *equality* capable of proving both *parametricity theorems* and *extensionality principles* for types, as these are both necessary for the internalisation of proofs such as Wadler's. My investigations into type-theoretic notions of equality for this purpose ultimately led me to *Cubical Type Theory*, which arose out of efforts to give a computational interpretation to the theory of *Homotopy Type Theory* (HoTT) pioneered by Voevodsky et al. As McBride has remarked, it is 'inspiringly simple' to prove principles such as function extensionality within Cubical Type Theory.[4] Indeed, an initial motivation for Cubical Type Theory, and HoTT more generally, was precisely the inability of the usual formulation of equality in Martin-Löf-style type theories to prove such extensionality principles while maintaining decidability of type checking.

[4] in a talk at the EUTYPES 2018 conference: https://youtu.be/W5-ulP_JzNc

However, as a relatively young branch of type theory, Cubical Type Theory has its share of technical problems that remain to be worked out. A particular snag is that by default, the extensionality principles provable within Cubical Type Theory only apply to terms in η-long form. In order to generalise these principles to *all* terms, the theory must be equipped with some form of η-conversion. Traditionally, including a form of either η-reduction or η-expansion in the notion of computation of a type theory is difficult to get right, and complicates the proofs of most metatheoretic properties. Extant cubical theories have mainly opted instead to either implement a separate notion of *judgemental equality* on top of computation (as in the case of Chapman, Nordvall Forsberg, and McBride) that implements η-conversion, or to forego a primitive notion of computation altogether, and to only rely on such *judgmental equality* (as in the case of Cohen, Coquand, Huber & Mörtberg). The downside of this approach is that it is generally more difficult to show that *judgmental equality* is decidable than it is to prove decidability of a notion of *computation*; the latter follows from normalisation and confluence, while to prove the former, one usually ends up defining a notion of computation anyway and showing that the judgmental equality is equivalent to it. For these reasons, I found it necessary to bite the bullet and work out the technical details of a type theory with a *computational* notion of η-conversion, before proceeding to extend this to a *cubical* theory.

Happily, many of the technical problems posed by the implementation of computational η-conversion were greatly simplified for me by the bidirectional style I had adopted in prior stages. A perennial question in type theory asks whether it is better to compute η-conversion by way of *η-reduction* or *η-expansion.* The bidirectional discipline settles this dispute in favour of η-expansion. The basic bidirectional theory developed at the first stage allowed a type-*checkable* term to be converted into a type-*synthable* term by annotating the term with its type; this had the consequence that every β-redex was labelled with its type. If we dualise this construction, and allow type-*synthable* terms to be similarly labelled with their types, the terms labelled in this way are just those to which η-expansion can be applied. This preserves the invariant that every site of active computation is labelled with the type at which it is active. As a result, the proofs of normalisation and confluence developed at the first stage go through without any trouble. The proof of subject reduction requires a few additional lemmas to cover cases introduced by η-expansion, but these are all proved by relatively painless inductions on derivations. The proofs of canonicity and decidability proceed virtually unchanged – the only difference is that now *both* of the rules for annotating checkable and synthable terms with their types are shown to be eliminable from the theory. This is an instance of the bidirectional discipline's close connection to Gentzen's sequent calculus – the fact that β-reduction allows elimination of the annotation rule for checkable terms corresponds to *cut elimination*; that η-expansion allows elimination of the annotation rule for synthable terms corresponds to *identity elimination.*

This suffices to show that both the basic type theory developed at the first stage, and the framework of QTSs developed at the second stage, can be extended with η-expansion while preserving all the **key properties** of these theories. However, one drawback of this approach to η-expansion is that it renders the theory more *verbose* than those of prior stages – η-expansion involves labelling terms whose types may be *synthesised* with their types, but because these types could already be *synthesised*, for the purposes of *type checking*, such annotations are redundant. This is one instance in which the duality between checking and synthesis in the bidirectional style does not *quite* match the symmetries of sequent calculus, despite being closely connected to them. At first this was cause for some annoyance on my part, but I quickly found a way to overcome this nuisance. Since the bidirectional discipline guarantees that an *unlabelled* site of η-expansion has enough information present within it to derive its type (and therefore a label for it), we can have our cake and eat it too: we can *write* programs without labelling sites of η-expansion with their types, and then translate these into *fully labelled* programs for the purpose of computation.

Fourth stage – propositional extensionality: Cubical Type Theory

> The history of typed programming is the history of the struggle for equality – Conor McBride[5]

Considerations of η-conversion out of the way, the final stage of development for the type theory involved combining the concept of *parametric quantification* developed in the framework of QTSs with a form of propositional equality derived from Cubical Type Theory. My main references for the construction of the theory at this stage, were Cohen, Coquand, Huber & Mörtberg (CCHM, hereafter)[11] and Chapman, Nordvall Forsberg & McBride (CNFM, hereafter).[12] In order to better acquaint myself with the background and motivation for the constructs of Cubical Type Theory, I made a significant study of Homotopy Type Theory.[13] The notions of equality found in both Cubical Type Theory and Homotopy Type Theory differ markedly from more ordinary conceptions of equality in logic, set theory, etc., and potentially have profound ramifications for logic, computation, and mathematics.

Voevodsky's key insight was to observe that a proof of the identity of two elements a and b of type A could be thought of as a *path* from a to b in the *space* defined by A. Under this view, every type has the structure of a space in homotopy theory. Moreover, since there is a type of identity-proofs $a =_A b$, one can consider paths between elements of *this* type, i.e. paths between paths, giving rise to complex higher-dimensional structures. Reasoning about the equality of terms thereby becomes 'type theory in n dimensions', and takes on a markedly *geometric* flavour. Cubical Type Theory attempts to give a computational interpretation to such higher-dimensional reasoning. The formalisation of equality proofs in Cubical Type Theory is striking in its simplicity – a proof of the equality of a and b is a function f whose argument ranges over a special type (or, rather, *pre-type*) $\mathbb{I}$, called the *interval*, with two designated 'endpoints' $0 : \mathbb{I}$ and $1 : \mathbb{I}$, such that f yields a when applied to 0, and b when applied to 1. The theory must then be so-constructed as to ensure that the elements of the interval are *indistinguishable* within the theory, so that any outputs of such a function must also be so-indistinguishable. This licenses treating such functions as proofs of *equality.*

Systems of Cubical Type Theory such as CCHM usually contain additional constructs for interpreting other aspects of Homotopy Type Theory, namely Voevodsky's Univalence Axiom and Higher Inductive Types. However, Cubical Type Theory's formalisation of identity proofs on its own is quite powerful, and merits consideration as an answer to the question of what constitutes a *constructive* proof of equality, independent of other aspects of the theory. Notably, most formulations of the Brouwer-Heyting-Kolmogorov (BHK) interpretation of intuitionistic logic (a standard formulation of constructive proof, and precursor to the Curry-Howard correspondence) conspicuously lack a clause for constructive proofs of identity. The approach to such proofs taken by Cubical Type Theory suggests that we may augment BHK with the following:

- a proof of $a = b$ is a function whose argument ranges over the interval $\mathbb{I}$, that yields a when applied to 0, and b when applied to 1, and that *never* uses its argument in a way that is relevant to the value returned by the function.

This suggests a strategy for unifying the treatment of equality in Cubical Type Theory

[5]quote from the following post on McBride's blog: https://pigworker.wordpress.com/2015/01/06/observational-type-theory-the-motivation/

with the notion of *parametric quantification* developed in the framework of QTSs previously: since terms marked with the quantity 0 in a QTS are *value-irrelevant*, this quantity can be used to mark *both* parametrically quantified variables, *and* the variables introduced by proofs of equality. This allows an alternative treatment of the interval in the theory I have developed. In most extant systems of Cubical Type Theory, the interval must be kept separate from other types, and indeed cannot be treated *as* a type within the theory, since the manner of quantification over elements of the interval differs significantly from that for any other type. The inclusion of a parametric modality of quantification reveals that the interval as used in proofs of identity is in fact just the image of an appropriately-structured type under *parametric* quantification. Rather than taking the parametrically-quantified interval as primitive, and thereby being forced to treat it separately from other types, I can instead leverage continuous quantification over such a type to get the intended behaviour of identity proofs.

As expected, within the system of Cubical Type Theory I have developed, extensionality principles are provable for all types, as are other desirable properties e.g. contractibility of singletons.

From here, somewhat surprisingly, only slight additions to the theory are needed in order to make parametricity theorems internally derivable. The main sources I consulted in studying the internalisation of parametricity in type theory were Bernardy, Moulin & Coquand (BCM),[14] and Harper & Cavallo (H&C);[15] however, the approach I have ultimately taken toward such internalisation diverges significantly from that of either set of authors, and to my knowledge is largely original. I make use of the fact that, within the theory I have developed, both *parametric* and *value-relevant* quantification over the interval are possible. This allows me to introduce a new construction on types: given types S, T and a type expression R with variables x (of type S) and y (of type T) free (which is to say, R is a *relation* on S and T), and elements i, j of the interval, one may form the type $(x : S) \times^i_j (y : T).R$, which I call a *graph type.* If $i = 1$ and $j = 0$, a graph type of this form reduces to the type S, while if $i = 0$ and $j = 1$, then the graph type reduces to T. If both $i = 1$ and $j = 1$, then the graph type reduces to the type of dependent triples (s, t, r) such that r is a proof of R with s substituted for x and t substituted for y (i.e. the *graph* of R). The terms i, j are then (clearly) *relevant* to the value of such a graph type. However, whenever a type of this form gets used in a *parametric* position, i and j also become parametric. This yields a very direct method for showing that parametric quantification preserves relations on types, and so in this sense captures Reynolds' idea of *relational parametricity.* Because this approach to internalising parametricity differs so markedly from those of BCM and H&C, I am not at all sure what the relationship is between these approaches (e.g. whether or not they are equivalent). Nonetheless, as detailed in the next section, this augmentation of the theory yields all the expected parametricity theorems, at least for the cases I have so far investigated.

As to proofs of the **key properties**, with some care these can all still be extended to the system of cubical type theory I have developed. This is particularly the case because the system I have developed does *not* yet include constructs for interpreting the univalence axiom. Further work will be necessary to determine whether it is possible to extend the type theory I have developed to a univalent system while preserving the **key properties**. Setting this problem aside for future research, I turned my attention to the task of showing that the desired parametricity theorems are all provable within the theory so-far developed.

Theorems for free!

I now present a catalogue of the various *free theorems* I was able to derive within the type theory built up over the previous stages. In order to prove most of these theorems, it was necessary to work within the *impredicative* form of the theory, due to the reliance on Church encodings:

- As anticipated, since function extensionality and parametricity principles for the relevant types are derivable within the theory, I was able to internally recreate Wadler's proof of induction for the type of Church numerals.
- As a generalisation of the above, I was able to show that the expected induction principles are provable for the Church-encodings of *all* strictly-positive inductive types within the theory.
- As a further generalisation, I was able to show that inductive-inductive types admit a representation in terms of Church encodings, for which induction principles are again provable.
- I was able to model some higher-inductive types (e.g. the circle, the torus) via Church-encodings, and prove that these encodings satisfy the expected *induction* and *path induction* principles. Unlike in the case of strictly positive inductive types, there is as yet no precise characterisation of what higher-inductive types are *in general*, so there is no immediate way of showing that *all* higher inductive types are so-representable within the theory.

These results serve to confirm the hypothesis that initially motivated this project: that a system combining parametric quantification with cubical path types would be capable of internally proving parametricity and hence also *induction* principles for Church-encodings of mathematical structures, and therefore could provide a valuable test-bed for experimenting with the type-theoretic treatment of such structures. On this basis, I consider the project to be a resounding success.

Récoltes et semailles

The work I undertook for this project proved to be incredibly edifying. I found a visceral satisfaction in seeing pieces I had arranged at the earlier stages of development come beautifully together later on. External circumstances rendered life difficult for me during the period of my research, and I was beset by personal problems and poor health, both mental and physical; in spite of all this, I am quite proud of what I have managed to achieve with this project, and very much hope to be able to do similar work in the future. The good news, in this regard, is that so much work remains to be done! At present, outside of this report, most of my writings on the topics mentioned herein are scattered throughout various notes. Further work will be required to collate and flesh out the proofs and ideas I have *sketched* into a comprehensive and formal presentation.

Beyond collecting and refining the results I have established so far, there are many open problems pertaining to the theory I have developed, its semantics, its applications, etc., some of which I have noted as they have arisen in prior sections of this report; investigations into these questions will likely lead to further refinements of and modifications to the theory as a whole. Additionally, the research I have done in service of this project has broadened my horizons to new questions, and – potentially – other projects. Suffice it to say that these and related problems form prime candidates for the subjects of future projects, dissertations and/or theses I will undertake as part of my academic career.

Prior to undertaking this project, I had an interest in becoming at least conversant in higher mathematical subjects such as category theory, algebraic topology, homotopy theory, and – of course – homotopy type theory. Among the greatest impacts this project had upon me, the research I conducted into these and related subjects over the Summer transformed this interest of mine into a fascination and deep desire to learn more. I have come to the conclusion that, in the past several decades, there has been something of a silent revolution in both logic and computation, spurred by increasing awareness of connections between the two, and that this revolution has been made possible, at least in part, by advances in the above mathematical disciplines. In my opinion, these connections between logic, computation, and mathematics yield profound conceptual tools, new forms of understanding, and novel approaches to problem-solving, all of which I seek to better understand, and, hopefully, contribute to. Indeed, I could see myself becoming something of an evangelist for what I take to be the new forms of reasoning fostered by the awareness of such connections. In order to gain a more complete mastery of these connections myself, I am now determined to have a better mastery of higher mathematics. Because my degree is technically in subjects outside of mathematics, my options for studying such mathematical subjects in a formal capacity are currently somewhat limited. In the coming months/year, I will be considering and discussing what options I have with my tutors, in order to determine the best way to proceed with a study of such higher mathematics, whether by way of independent study, or perhaps by enrolling in a graduate program with a mathematics component.

Ultimately, I see the investigations I have undertaken in this project as a stepping-stone along the path to a more general theory of abstraction. Ideally, such a theory would provide a standard by which to assess the *means of abstraction* afforded by various programming languages, mathematical theories, and logical systems, and also to guide the design of new such systems. In particular, it is my hope that the type theory I have developed toward this end (or something like it) could serve as the kernel of a system that is at once programming language and proof assistant/automated theorem prover (in a similar manner to how the Calculus of Inductive Constructions forms the kernel of the Coq proof assistant, and Martin–Löf's Intuitionistic Type Theory forms the basis of the Agda programming language, etc.)

In order to be able to conduct more investigations along these lines, an academic research career seems appropriate for me, and so I intend to pursue a PhD. However, I remain open to other options that would similarly allow me to undertake such research. For instance, if I am able to develop a programming language/proof assistant on the basis of these logical investigations that becomes widely used enough that some organisation (whose ethics I can tolerate) is willing to sponsor me in the development of this language, then I could see that also being a viable career path that would hopefully permit me the time to conduct my own research, etc. Needless to say, this project has provided me with experience that will be incredibly valuable, whatever my path ends up being. I am extraordinarily grateful and honoured for having received the college's support in this project. Additionally, I am grateful to my Merton tutors, Luke Ong and Simon Saunders, for having supported my application and provided feedback on my project proposal, as well as resources to explore in the course of my project.

What I have managed to show within the theory I have so-far developed has convinced me, and I hope will be convincing to others, that the task of developing a theory of abstraction is a worthy one, and while perhaps too broad to ever be fully accomplished, may nonetheless bear worthwhile fruit. The problem of finding the appropriate level of generality from which to view generality itself may appear impenetrable, like some great mass of land that looms far above the waters of our understanding; yet the sea rises on.

C.B. Aberlé, Oxford, 24 September 2020

References

1. H. Abelson, G. Sussman & J. Sussman (1996), *The Structure and Interpretation of Computer Programs*, Cambridge, MA: MIT Press, 2nd ed.
2. J. Reynolds (1983), "Types, Abstraction and Parametric Polymorphism", Information Processing 83: 513-524
3. P. Wadler (1989), "Theorems for free!", *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pp. 347-359
4. P. Wadler (2007), "The Girard-Reynolds Isomorphism", 2nd ed., Theoretical Computer Science 375: 201-226

Bidirectional type checking

5. C. McBride (2018), "Basics of bidirectionalism", published to his personal website: https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectio nalism/
6. J. Dunfield & N. Krishnaswami (2019), "Bidirectional Typing", retrieved at: https://www.cl.cam.ac.uk/~nk480/bidir-survey.pdf

Normalisation arguments for dependent type theory

7. C. Casinghino (2010), "Strong Normalization for the Calculus of Constructions", retrieved at: https://prosecco.gforge.inria.fr/personal/hritcu/temp/snforcc.pd f

Quantitative Type Systems

8. R.N. Mishra-Linger (2008), "Irrelevance, Polymorphism, and Erasure in Type Theory", *Dissertations and Theses*, paper 2674
9. C. McBride (2016), "I Got Plenty O' Nuttin", In S. Lindley, C. McBride, P. Trinder & D. Sannella (Eds.): *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Springer, 207–233
10. R. Atkey (2018), "The Syntax and Semantics of Quantitative Type Theory", in *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom*

Cubical Type Theory

11. C. Cohen, T. Coquand, S. Huber, A. Mörtberg (2016), "Cubical Type Theory: a constructive interpretation of the univalence axiom": arXiv:1611.02108
12. J. Chapman, F. Nordvall Forsberg, C. McBride (2018), "The Box of Delights (Cubical Observational Type Theory)", unpublished note, retrieved at: https: //github.com/msp-strath/platypus/blob/master/January18/doc/CubicalO TT/CubicalOTT.pdf

Homotopy Type Theory

13. Univalent Foundations Project (2013), *Homotopy Type Theory – Univalent Foundations of Mathematics*, Princeton: Institute for Advanced Study

Internal Parametricity

14. G. Moulin (2016), “Internalizing Parametricity”, *thesis based on prior work of*:
    1. J.-P. Bernardy & G. Moulin (2012), “A computational interpretation of parametricity”, *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science*, pp. 135–144
    2. J.-P. Bernardy & G. Moulin (2013), “Type theory in color”, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP ’13*, pp. 61–72
    3. J.-P. Bernardy, G. Moulin & T. Coquand (2015), “A presheaf model of parametric type theory”, *The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, vol. 319, pp. 67–82
15. R. Harper & E. Cavallo (2020), “Internal Parametricity for Cubical Type Theory”, retrieved at: http://www.cs.cmu.edu/~rwh/papers/bridges/csl20.pdf