Thesis for The Degree of Doctor of Philosophy

On Induction, Coinduction and Equality in Martin-Löf and Homotopy Type Theory

Andrea Vezzosi

Division of Logic and Types
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2018

On Induction, Coinduction and Equality in Martin-Löf and Homotopy Type Theory

Andrea Vezzosi

ISBN 978-91-7597-772-0
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 4453.
ISSN 0346-718X

Technical Report No 160D
Department of Computer Science & Engineering
Division of Logic and Types
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2018.

Abstract

Martin Löf Type Theory, having put computation at the center of logical reasoning, has been shown to be an effective foundation for proof assistants, with applications both in computer science and constructive mathematics. One ambition though is for MLTT to also double as a practical general purpose programming language. Datatypes in type theory come with an induction or coinduction principle which gives a precise and concise specification of their interface. However, such principles can interfere with how we would like to express our programs. In this thesis, we investigate more flexible alternatives to direct uses of the (co)induction principles.

As a first contribution, we consider the n-truncation of a type in Homotopy Type Theory. We derive in HoTT an eliminator into (n+1)-truncated types instead of n-truncated ones, assuming extra conditions on the underlying function.

As a second contribution, we improve on type-based criteria for termination and productivity. By augmenting the types with well-foundedness information, such criteria allow function definitions in a style closer to general recursion. We consider two criteria: guarded types, and sized types.

Guarded types introduce a modality "later" to guard the availability of recursive calls provided by a general fixed-point combinator. In Guarded Cubical Type Theory we equip the fixed-point combinator with a propositional equality to its one-step unfolding, instead of a definitional equality that would break normalization. The notion of path from Cubical Type Theory allows us to do so without losing canonicity or decidability of conversion.

Sized types, on the other hand, explicitly index datatypes with size bounds on the height or depth of their elements. The sizes however can get in the way of the reasoning principles we expect. Our approach is to introduce new quantifiers for "irrelevant" size quantification. We present a type theory with parametric quantifiers where irrelevance arises as a "free theorem". We also develop a conversion checking algorithm for a more specific theory where the new quantifiers are restricted to sizes.

Finally, our third contribution is about the operational semantics of type theory. For the extensions above we would like to devise a practical conversion checking algorithm suitable for integration into a proof assistant. We formalized the correctness of such an algorithm for a small but challenging core calculus, proving that conversion is decidable. We expect this development to form a good basis to verify more complex theories.

The ideas discussed in this thesis are already influencing the development of Agda, a proof assistant based on type theory.

Keywords

Type Theory, Higher Inductive Types, Guarded Types, Sized Types, Parametricity, Conversion

Acknowledgment

I want to thank my supervisor Andreas Abel for his curiosity and confidence which supported my studies for all these years. I am also grateful to Nils Anders Danielsson for being my co-supervisor and before that for pushing me to apply as a PhD student, which I might have thought out of reach.

My thanks to Rasmus Møgelberg for accepting the role of opponent, and to the members of my grading committee and backup: Robert Atkey, Jean-Philippe Bernardy, Hugo Herbelin, Andrew Pitts.

I want to thank Simonn Huber, Víctor López Juan, Bassel Mannaa, Jesper Cockx, Fancesco Mazzoli, Guilhem Moulin, Anders Mörtberg, Fabian Ruch and Christian Sattler for sharing their time as students (or post-) at Chalmers with me. And all of the Logic and Types group for the opportunity to work with succh stimulating people. I am also grateful to all my other colleagues and friends at the department for providing such a welcoming environment to work in.

Special thanks to my wife for all her support and for making me move many commas two words to the right.

Contents

Chapter 1

Introduction

Martin-Löf Type Theory (MLTT) [30] has been very successful as the foundation of proof assistants capable of formalizing large mathematical proofs like the *Four Color Theorem* [20] and the *Feit-Thompson Theorem* [21] but also results in computer science like correctness of a C compiler [27].

MLTT can also be used directly as an expressive type system for a functional programming language. Types then do not just express correctness properties to verify after the fact but also guide development by constraining the possible programs towards the correct one, as for example when implementing search trees [31]. Of course, this guidance can also become a burden when the constraints are cumbersome to satisfy. Even when keeping the types very simple, a user has to write a program that the theory can recognize as total, which is something that mainstream programming languages do not request. The burden is significant enough that implementations like Agda and Idris provide pragmas to circumvent it and instead accept the programmer's judgment and/or mark the definition as untrusted.

A core calculus would typically ensure totality by providing only (co)induction combinators, which also have the benefit of being easy to model and fit well within categorical semantics as universal properties. They correspond to primitive (co)recursion or (un)folds, which are not that expressive in practice, as witnessed by the quest for powerful generalizations [22]. The state of the art in proof assistants based on type theory is instead to allow pattern matching and direct recursion, and deploy more or less sophisticated coverage and termination checks. However these are fairly limiting, especially because they do not allow the programmer to provide their own reasoning to convince the checker.

1.1 This Thesis

This thesis is a collection of six papers divided into three parts. The first deals with the induction principle of n-truncations in HoTT. The second with guarded types and sized types as type-based criteria for termination and productivity. The third with decidability of conversion of type theory.

1.1.1 First Part

The expressivity issues with writing programs discussed so far also apply to what might seem closer to formalizations of mathematics, as long as they involve non-trivial constructions. Homotopy Type Theory [39] is a field that connects homotopy theory and type theory. The connection centers around the identity type, whose elements can be thought of as paths connecting the two values being equated. Types are then interpreted as topological spaces up to homotopy equivalence.

It then becomes natural to classify types according to the complexity of the topology of their associated space, and we can do so by definitions completely internal to the theory, rather than refer to the interpretation: we call a type *contractible* if it is equivalent to the unit type, then we say that a type has homotopy level n (or is an n-type) if its $(n+2)$-iterated identity type is contractible. A -2-type is then a contractible one. As an example, 0-types behave much like discrete spaces since all paths are trivial, as in any two parallel ones are equal, and so they are a suitable representation of sets [38]. Instead, (-1)-types are regarded as mere propositions, i.e., proof-irrelevant, meaning that all elements of a -1-type are connected by a path and it only matters whether there is some element or none at all. It happens that some constructions, like pushouts, do not naturally preserve the homotopy level, e.g., build sets out of sets. Or maybe we would like to turn constructive existence into a mere proposition. For these and other reasons it can be useful to truncate a type to a desired homotopy level. Given an arbitrary type A, its n-truncation $\|A\|_n$ is the least n-type with a map $|-|_n : A \to \|A\|_n$. Equivalently, its standard induction principle says that a map $\|A\|_n \to B$ can be built from a map $A \to B$ as long as B is also an n-type. In the paper "Functions out of Higher Truncations" we relax this to B being an $(n+1)$-type as long as the function is constant on all $(n+1)$ loop spaces, which we call n-constant. If we think in terms of cubes, the m iterated loop space of a type at some point a is the space of m-cubes whose $m-1$-dimensional faces are degenerate, in the sense that they are actually the constant cube at a. The paper gives two proofs, one based on a reformulation of $\|A\|_n$ as an $(n+1)$-truncated Higher Inductive Type (HIT) [39] with extra constructors, the other on the n-connected components of the type A:

$$\mathsf{conn}_n(x : \|A\|_n) := \Sigma(a : A).\, |a|_n = x$$

By contractibility of singletons we have the following equivalence

$$A \simeq \Sigma(x : \|A\|_n).\, \mathsf{conn}_n(x)$$

from which we can show that an n-constant map $A \to B$ is a family of n-constant maps $f_x : \mathsf{conn}_n(x) \to B$. Each of the n-constant f_x is actually determined by a single point in B, which is not surprising since collectively they are supposed to correspond to a map from $\|A\|_n$ to B. This motivates us to prove the following family of equivalences,

$$\Pi\, x : \|A\|_n.\, (B \simeq \Sigma(f_x : \mathsf{conn}_n(x) \to B).\, n\text{-}\mathsf{constant}(f_x))$$

and we do so by induction on n and by the fact that being an equivalence is a (-1)-type. The latter allows induction on x so that we can define the

inverse function while assuming $x := |a|_n$ for some $a : A$. We can then collect this family of equivalences into an equivalence of families, which concludes the result.

This extended elimination principle has been used in "Constructions with Non-Recursive Higher Inductive Types" [25] to give a definition of propositional truncation without making use of recursive higher inductive types. The same paper provides alternative elimination principles for n-truncations, but while they allow the codomain to be an $(n+k)$-type for an arbitrary k, they impose stronger than expected constraints on the map $A \to B$, so there is still further work to be done in this direction.

1.1.2 Second Part

The second part of this thesis presents some contributions to type-based totality checking, which in our case refers to systems with sized types [24] or guarded recursive types [32].

Programming in a functional language typically involves (co)recursive definitions dealing with algebraic datatypes. In dependent type theory, admitting only total definitions is a prerequisite for consistency, as otherwise the empty type could be inhabited. Therefore, typical implementations which allow direct (co)recursion employ termination checkers based on a structural ordering [19], possibly with some extensions [1, 26, 44]. Such termination checkers, however, get in the way of some typical functional programming patterns like the use of higher order functions. We can look at a Haskell example that defines a map function for rose trees.

```
data RT a = Node a [RT a]

mapRT :: (a -> b) -> RT a -> RT b
mapRT f (Node x ts) = Node (f x) (map (mapRT f) ts)
```

If we were to translate `mapRT` to Agda, the termination checker would complain that the recursive call is not justified because it is not visibly applied to any subterm of `Node x ts`, and in fact whether this function is total depends on the behaviour of `map`. Here, a simple workaround is to mutually define a specialized `map-mapRT` so that the recursion pattern becomes explicit. This kind of inlining is not always possible, though. As another basic example, let us take `unfold`.

```
unfold :: (a -> Maybe (a, b)) -> a -> [b]
unfold f x = case f x of
  Just (x', b) -> b : unfold f x'
  Nothing      -> []
```

If we consider `[b]` as the type of finite lists, then whether `unfold f` terminates depends on the behaviour of `f`, which in this case is unknown until `unfold` gets applied.

Type-based totality checking instead aims to keep track of the relevant information about functions' behaviour in their type. A major component is to internalize the fact that (co)inductive types are typically given semantics as the point where a chain of approximations to the type stabilize. If the

language can discuss the intermediate steps in those chains then it has more expressivity to discuss the information flow of recursive definitions.

This approach is very explicit with sized types. For example, in Agda we could define a sized version of the type of rose trees, `RT A i`, which is inhabited by trees of height bounded by the size `i`, then the definition of `mapRT` would be accepted as total because the recursive call would be at type `RT A j → RT B j` for some size `j` smaller than `i`. Guarded recursive types, instead, keep this staging more implicit by the use of Nakano's later modality, $\triangleright$, which is used to "guard" recursive positions in the definition of types like `Stream`, so that types can keep track of the ability to access what, in this case, would be the tail. Other than its applications to productivity of corecursive and "tying-the-knot" definitions [10], a motivating application of guarded types comes from the observation that it can be useful to consider chains of types that never actually stabilize. This happens with recursive types which appear in their definition in a negative position, as for example a type T such that $T \simeq (\triangleright T \to A)$. Such types have been used to formulate models of program logics as a less bureaucratic alternative to explicit step-indexing [14].

Guarded Types

In order to integrate the $\triangleright$ modality into intensional type theory it is necessary to develop its operational semantics, even in the presence of free variables, so that we can formulate a useful but decidable judgmental equality. The following two papers confront the challenge to unrestricted computation that the guarded fixed point poses.

A Formalized Proof of Strong Normalization for Guarded Recursive Types studies a simply typed lambda calculus with products extended with guarded recursive types and the applicative functor structure of the $\triangleright$ modality:

$$\mathsf{next} : A \to \triangleright A$$
$$\star : \triangleright(A \to B) \to \triangleright A \to \triangleright B$$

A guarded fixed point combinator $\mathsf{fix} : (\triangleright A \to A) \to A$ can be derived using the recursive type $T = \triangleright T \to A$.

Full reduction for this language is the congruence closure of the usual β rules for application and projections plus the law $(\mathsf{next}\, t) \star (\mathsf{next}\, u) \mapsto \mathsf{next}(t\, u)$. Full reduction invalidates strong normalization as it leads, for instance, to the infinite chain $\mathsf{fix}\, f \to f\, (\mathsf{next}(\mathsf{fix}\, f)) \to f\, (\mathsf{next}(f\, (\mathsf{next}(\mathsf{fix}\, f)))) \to \ldots$ As a remedy, we lift the concept of finite approximations directly from the model and define a reduction relation indexed by a natural number n, which specifies under how many nested uses of next we are still allowed to reduce. This strategy takes inspiration from the denotational semantics: if a term is supposed to represent a natural transformation in the topos of trees, $\mathsf{Set}^{\omega^{\mathsf{op}}}$, then we are entitled to apply it to a natural number and observe its behaviour at that stage. However, the semantics of next ignores its argument at stage zero, and it is this base case that bootstraps the recursion that justifies fix, so it seems natural for reduction to also stop there.

The proof itself goes by showing that saturated sets [11], indexed by the stage as a natural number, form a model of the calculus. By saturated set we

mean a predicate over terms which sits between the strongly neutral and the strongly normalizing terms and, moreover, supports renaming and is closed under strong head expansion. Indexing by the stage is necessary because our notions of neutral, normal, and expansion inherit the stage from the reduction strategy indicated above. Eventually, we prove that the predicates corresponding to the various type formers are antitone with respect to the stage, so it might be worthwhile to reformulate the model as predicates internal to the category of presheaves over typing contexts and stages.

The fact that a term might have a different normal form for each stage introduces some problems for using this as a strategy to decide equality, though, because later stages will generally recognize more terms as reducing to the same normal form. For base types and with a restricted context the stage does not matter, but in general we do not provide a bound on which stages to try when comparing two terms. An option would be to design a type theory where the staging is expected to influence the judgmental equality: one possibility is to consider the stage as a fuel we earn by walking down our terms during typechecking and then get to spend in the conversion rule when comparing equality of types. We would have a typing judgment of the form $\Gamma \vdash_n t : A$, where the stage would increase under a next

$$\frac{\Gamma \vdash_{n+1} t : A}{\Gamma \vdash_n \mathsf{next}\, t : \triangleright A}$$

and then used by conversion:

$$\frac{\Gamma \vdash_n t = u : A}{\Gamma \vdash_{n+1} \mathsf{next}\, t = \mathsf{next}\, u : \triangleright A} \qquad \frac{}{\Gamma \vdash_0 \mathsf{next}\, t = \mathsf{next}\, t : \triangleright A}$$

Top-level definitions would then be checked at stage zero. The usefulness of such a system and whether it enjoys the usual expected meta-theoretic properties could be interesting future work.

Finally, since the strong normalization proof is fully formalized in Agda, it has been stripped of the later modality components and taken as the prototype implementation for the POPLMark Reloaded challenge [35], which aims to compare different approaches to syntax with binding in proof assistants by how they apply to a fairly involved but well-understood proof like strong normalization for the simply typed lambda calculus.

Guarded Cubical Type Theory: Path Equality for Guarded Recursion presents Guarded Cubical Type Theory (GCTT) as an intensional version of the extensional Guarded Dependent Type Theory (GDTT) [15].

As the name suggests GDTT is a dependent type theory, including Π and Σ types, a natural number type, an equality type, and a hierarchy of universes, extended with the $\triangleright$ modality and clock quantification. The main innovation is the introduction of delayed substitutions in place of the $\star$ operator: they appear as extra arguments of next and $\triangleright$ and allow the formation of types like $A : \mathbb{N} \to \mathsf{U}, x : \triangleright \mathbb{N} \vdash \triangleright[n \leftarrow x].\, A\, n$. GDTT furthermore makes fix a primitive, since in the presence of a universe it can be used to derive guarded recursion in types, instead of the other way around. The theory is motivated by a few

examples of non-trivial corecursion patterns and proofs of equality of guarded streams by bisimulation.

Judgmental equality is undecidable not only because of the reflection rule, but also due to the rule for unfolding fix, $\Gamma \vdash \mathsf{fix}\, f = f\,(\mathsf{next}(\mathsf{fix}\, f))$, which introduces the same normalization problems as discussed above. Simply removing the reflection rule and turning the unfolding of fix into a propositional equality would likely recover decidability but destroy canonicity, as now there would be proofs of equality that are not given by reflexivity and the eliminator for the equality type would be stuck on them.

This is a problem that arises whenever we start from an intensional type theory where propositional equality is seen as an inductive type with reflexivity as the only constructor and correspondingly a single β rule for the eliminator: canonicity dictates that in a closed context propositional equality and judgmental equality must coincide, so decidability of the latter also severely limits the former. Breaking this coincidence was the main feature of Observational Type Theory [7] which got around it by internalizing the setoid model of type theory and defining propositional equality on a type by type basis and, by doing so, achieving support for function and proposition extensionality[1].

Cubical Type Theory (CTT) [17] takes the intuitions of Homotopy Type Theory about the identity type a step further and uses paths as the representation of equality proofs themselves, as in maps from an abstract interval type: elements of $\mathsf{Path}_A\ a\ b$ are built using path abstraction $\langle i \rangle t$ where t is an element of A with an extra interval variable i and furthermore $t[0/i] = a$ and $t[1/i] = b$, where 0 and 1 are the two endpoints of the interval. Then, reflexivity is not the only canonical element of the equality type, function extensionality just follows from abstraction for paths and function types, and univalence can be proven with the support of a special Glue type that allows to build paths in the universe from equivalences. This does not come for free, as now the eliminator for paths has to deal with more canonical forms and its behaviour is defined according to the type we are eliminating into.

We obtain GCTT by extending CTT with the $\triangleright$ modality and next as in GDTT and adding a path dfix

$$\frac{\Gamma \vdash f : \triangleright A \to A \quad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \mathsf{dfix}\, f\, r : \triangleright A} \qquad \Gamma \vdash \mathsf{dfix}\, f\, 1 = \mathsf{next}\,(f\,(\mathsf{dfix}\, f\, 0)) : \triangleright A$$

so that we can define $\mathsf{fix}\, f := f\,(\mathsf{dfix}\, f\, 0)$ and prove it equal to its one step unfolding by $\langle i \rangle f\,(\mathsf{dfix}\, f\, i)$. Choosing dfix as the primitive rather than fix avoids the introduction of new canonical forms at an arbitrary type A and confines them to $\triangleright A$ instead. Since GCTT does not have clock quantification there is no real eliminator for the $\triangleright$ modality so there's no issue with piling up more term constructors for it. However, the prototype implementation[2] has some limited support for clock quantification and we were able to make dfix step forward in a controlled fashion when implementing the computational behaviour of prev[3]:

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright A}{\Gamma \vdash_{\Delta} \mathsf{prev}\, \kappa.\, t : \forall \kappa.\, A} \qquad \mathsf{prev}\, \kappa.\ \mathsf{next}\, t = \Lambda \kappa.\, t$$

[1] Also expected to support quotient types and bisimilarity for coinductive types

[2] `https://github.com/hansbugge/cubicaltt/tree/gcubical`

[3] Here we give only a simplified type

In fact, we can simply reduce as if $\mathsf{dfix}\, f\, r = \mathsf{next}\, (f\, (\mathsf{dfix}\, f\, r))$ and let prev strip away the next.

$$(\mathsf{prev}\ \kappa.\ \mathsf{dfix}\, f\, r) = \Lambda\kappa.\, f\, (\mathsf{dfix}\, f\, r)$$

The paper provides a denotational model in presheaves over $\square \times \omega$ which proves the consistency of our theory. The category of cubes $\square$ is the one used for the denotational semantics of CTT, so our model can be thought of as staged cubical sets.

Coinductive Types in Cubical Type Theory

To be fair, if the aim is just to extend CTT with coinductive types whose equality is bisimulation, we do not need to introduce the $\triangleright$ modality. As I discuss in [43] , although only for the case of streams, cubical sets already support coinductive types in the sense of strict final coalgebras[4] and they can be shown to be fibrant, i.e. support the eliminator for the equality type. Then, we just need a theory that allows paths to be defined corecursively and, for that, copatterns seem to be a very good fit, as shown by this example in "Cubical Agda" [41]:

```
record Stream (A : Set) : Set where
  coinductive
  constructor _,_
  field
    head : A
    tail : Stream A

open Stream

map : ∀ {A B} → (A → B) → Stream A → Stream B
head (map f xs) = f (head xs)
tail (map f xs) = map f (tail xs)

map-id : ∀ {A} {xs : Stream A} → map (λ x → x) xs ≡ xs
head (map-id {xs = xs} i) = head xs
tail (map-id {xs = xs} i) = map-id {xs = tail xs} i
```

We have that `map-id {xs = xs}` is a path between streams, so it takes an interval variable `i` as argument, and then it defines a stream by giving its `head` and `tail` in such a way that they match the `head` and `tail` of the streams we are proving equal. For the `tail` case we proceed by corecursion, as we would in case we were defining a simple function, rather than a path. This use of corecursion is still justified in the denotational semantics because we model streams by taking a final coalgebra across arbitrary cubical sets, so the universal property gives us a map even from a coalgebra that has the interval type as a component.

Of course, the usual limitations of productivity checking apply, so integration with type based approaches is still important.

[4] Being a presheaf category they can be built as limits, level-wise

Sized Types

In comparison to guarded types, *sized types* mention the stages explicitly. Further, besides antitone dependency on the stage as for corecursion, sized types can be inductive and increase with the size. However, since sizes are then actual arguments of the constructors of inductive types we have the problem that they interfere with their equality. For example, sized natural numbers $\mathbb{N}\, i$ end up having not just a single "zero" element but one for each size j which is less than i.

This creates distinctions where we would like none, and prevents us from deriving the actual type of natural numbers from the sized variant because the multiple zeros would prevent it from being the fixed point of the $\mathsf{Maybe}\ X = \mathsf{Unit} + X$ functor, at least in the canonical way.

However, we also cannot disregard sizes completely when comparing terms for equality because the difference between, e.g., the types $\mathbb{N}\, i$ and $\mathbb{N}\, (i+1)$ is crucial to ensure the well-foundedness of programs. We then need a way to keep track of where size arguments can be ignored or, in other words, when functions use them in a uniform enough way to produce the same result for different sizes.

If we look at the semantics of (co)inductive types as (co)limits of chains, then the uniformity we would like to impose is naturality. But internalizing naturality directly would require us to keep track of the variance of size variables in types and into the realm of directed type theory, which is not well understood yet [28, 37]. However, the semantics of lambda calculus give us relational parametricity [36], a concept which also tries to characterize the well-behavedness or invariance of maps for some of their arguments, and also specializes to naturality in many cases. Due to this, parametric quanfication seems a good fit for our purposes.

Parametricity has been studied before in the context of MLTT, both in the sense of categorical models satisfying parametricity [8] and as extensions of type theory internalizing parametricity principles [12]. These works study how the Π type itself enjoys interesting parametricity properties, in case the domain has an interesting notion of relation. However, they are confronted with limitations with regards to trying to scale the identity extension lemma to universes. The identity extension lemma (IEL)[36], used to derive naturality from parametricity, implies that the relational interpretation of a closed type is just equality. However, when proving interesting properties about parametrically polymorphic functions, we rely on the relational interpretation of the domain to be coarser than equality, most notably for the universe but not only [9]. A specific example of this phenomenon is that we would like parametricity to imply that a function of type $f : (A : \mathsf{U}) \to F\, A \to G\, A$ is a natural transformation when F and G are functors. But if we have $\mathsf{U} : \mathsf{U}_1$ and take $F\, A = \mathsf{Unit}$, $G\, A = \mathsf{U}$, we can write $f = \lambda(A : \mathsf{U})\, (\_ : \mathsf{Unit}).\, A$ which does not fit into a naturality square. As a consequence we have that polymorphism-as-Π guarantees naturality when the codomain is "simple" enough to prevent the leak of information coming from elements of the domain type but does not scale to more general situations.

Parametric Quantifiers for Dependent Type Theory addresses this problem by presenting a type theory (ParamDTT) which introduces new quantifiers that restrict the use of the quantified variable through a system of modalities, thus preventing definitions like f above. With those quantifiers and a type Size which supports well-founded parametric and non-parametric induction, we are able to internally build indexed initial and final coalgebras for functors that commute with parametric Size quantification, which includes (finitary-branching) polynomial functors.

We give a denotational semantics in a presheaf model over a category of cubes generated by two distinct interval objects with a map between them: we use one interval to characterize equality, and the other parametric relatedness.

The theory does not admit canonicity because our version of identity extension is introduced as an axiom which is only validated in the model. The theory already uses Path-like types from [12] and Cubical Type Theory [17] to discuss how values are parametrically related, so we expect that the techniques of CTT can be adapted to give computational meaning to the axioms we introduced.

Even without canonicity we provide a prototype implementation [42] as a fork of Agda by adapting the present support for `--cubical` and an irrelevance modality. We found it quite helpful to experiment with the power of the theory and used it to formalize the examples in the paper. As a side-product, the effort spent adapting Agda to support parametricity as a modality was then reused to produce a prototype implementation[40] of crisp type theory [29], which seems to indicate the infrastrucure developed could be useful for other systems of modalities.

Normalization by Evaluation for Sized Dependent Types also presents a theory with two quantifiers, $\Pi\ i : \mathsf{Size}\,.$ and $\forall\ i : \mathsf{Size}\,.$ to distinguish between occurences of sizes that should or should not matter for equality, but in this case the judgmental equality. The technical contribution of the paper is then a Normalization by Evaluation (NbE) algorithm for conversion checking.

The main hurdle is how to reconcile typed judgmental equality with the irrelevant quantifier $\forall i.$, since given $f : \forall i : \mathsf{Size}\,.T\,i$ we would like $f\ i$ and $f\ j$ to be equal, but they would naturally live in different types $T\ i$ and $T\ j$.[5]

To get around this, we take the approach of making the size in an application like $f\ i$ to also be irrelevant for the typing[6]

$$\frac{\Gamma \vdash t : \forall i : \mathsf{Size}\,.T[i] \quad \Gamma^{\oplus} \vdash a, b : \mathsf{Size}}{\Gamma \vdash t\,\langle a \rangle : T[b]}$$

so that we can use different sizes in the equality rule for irrelevant application while maintaing the invariant that both sides have the same type.

$$\frac{\Gamma \vdash t = t' : \forall i : \mathsf{Size}\,.T[i] \quad \Gamma^{\oplus} \vdash a, a', b : \mathsf{Size}}{\Gamma \vdash t\,\langle a \rangle = t\,\langle a' \rangle : T[b]}$$

[5] In ParamDTT, the parametric quantifiers only give propositional equalities, and we have a notion of heterogeneous equality between elements of two parametrically related types

[6] $\Gamma^{\oplus}$ is like Γ but where irrelevant bindings are turned relevant

This setup is motivated by the desire to approximate curry-style equality for church-style terms: the sizes in the application are there as annotations for a typechecking algorithm, but they are actually ignored by the type theory itself.

Operationally, the reason we can be so liberal with sizes is that in our type theory they never affect the overall "shape" of a type since size expressions only appear in size applications and as indexes of datatypes, and there's no size-case. That means that the same type directed eta equality rules will apply to both $t = u : T[i := a]$ and $t = u : T[i := b]$.

We prove conversion decidable by presenting an NbE algorithm and its proofs of soundness and completeness. The algorithm includes the usual reflection and reification steps, which take a type to orchestrate expansion to η-long normal forms. The notion of type shapes comes back in the completeness proof, as the types provided to reflection and reification end up diverging from the type we want to compare the terms at. But all is well as long as the former types are the same approximate shape as the latter (defined formally in the paper) because they will trigger the same eta-expansions. We only have to generalize the corresponding theorems to get the induction through.

The use of NbE is a limitation we would like to overcome in future work. For reasons that will be discussed in the next section we would like obtain an incremental conversion checking algorithm based on reduction to weak normal forms instead, but in that case when comparing neutral irrelevant size applications we would need to figure out a suitable size b to use to instantiate the type of the result.

1.1.3 Third Part

Decidability of Conversion

The previous part of this thesis proposes extensions of type theory aimed at increasing its expressivity. Such extensions are shown consistent through various models, but we also want them to be practically implementable.

Proof assistants like Coq and Agda insist on decidable judgmental equality and normal forms for terms. This simplifies the implementation by being able to omit the storage not only of typing derivations but also type annotations in terms. That is because redexes can sometimes be typed by fairly different derivations if we don't have the necessary annotations, but normal forms do not have this problem because we can deploy a bidirectional typechecking algorithm and infer some of those annotations back as needed. Another advandage of normalization is that one can rely on it for techniques like small scale reflection, which make use of computation at the level of types to automatically simplify goals rather than having to produce a proof term that encodes each computation step.

When extending type theory one would like to prove that such nice properties are preserved and, ultimately, that a reasonable typechecking algorithm can be deployed. A big part is proving that there is an algorithm for deciding whether two terms are equal at a given type, i.e. conversion. It's not a straightforward task because the behaviour of types is complicated by universes allowing arbitrary reductions in them, and the equality of terms is affected by types through the eta rule for dependent functions.

If one only wants to show decidability, one option is to prove a normalization theorem using, for example, a normalization by evaluation technique as we did above. The decision procedure would then be to fully normalize each term, then compare the normal forms. However, that is not how conversion checking is implemented in, e.g., Agda or Mini-TT [33, 18], where the algorithm is instead to reduce to weak head normal form, then compare the heads, and then recurse in subterms. If normalization happens lazily then the two algorithms can actually have very similar executions, but in such a setup it might be hard to tell if we are normalizing more than necessary. Also, a practical typechecker probably wants to insert shortcuts like checking for syntactic equality to skip recursion on subterms when possible. Side-stepping normalization is especially useful in the presence of unification variables because the incremental comparison algorithm can be easily extended to the case where one or both of the terms is an application of an unification variable which can potentially be solved without further reduction.

Decidability of Type Theory in Type Theory , for the reasons above, takes the option of formalizing the correctness of a conversion checking algorithm based on typed weak head normalization and type-directed comparison. Specifically, we define algorithmic equality as an inductively defined relation, which we then show to be decidable and equivalent to judgmental equality.[7]

Our proof goes by a Kripke logical relation with contexts as worlds and renamings as morphisms between worlds. The relation is over open terms, hence the Kripke structure, and based on reduction in the sense that membership is determined by the weak head normal form (whnf) of a term. Due to this, we call it reducibility. For types the relation is defined inductively and recurses in the subterms of their whnf, for terms in a type it is defined by recursion on the proof that the type is reducible, and it checks that the observations that are possible for the whnf of the term are also reducible. By the same recursion we also define notions of reducible equality for types and terms. The fundamental theorem is proven as usual by quantifying over a reducible substitution[8] and by showing that the identity substitution is reducible. It is only after this proof that a lot of intuitions about the typing and equality judgments are validated: injectivity of type constructors and other inversion lemmas are not easy to establish from the declarative formulations of judgmental equality.

Establishing such properties is, however, not the whole battle. Earlier formulations of this proof [4, 2] relied on a second distinct logical relation with its own fundamental lemma to prove that algorithmic equality is complete, i.e. it is implied by judgmental equality. The two logical relations differ by which equality relation they imply through their "escape" lemmas, in one case judgmental and in the other algorithmic.

Using two logical relations and fundamental lemmas might be the most pragmatic presentation when doing things informally: most cases of the proof will be omitted anyway, and it would otherwise be hard to keep track of all the assumptions necessary if one tries to create an abstraction to reuse. However,

[7] Regarding the relationship with normalization: from a proof that a term is algorithmically equal to itself we can extract its normal form and a proof that it is judgmentally equal to the term.

[8] However, we package this quantification into its own relation, denoted $\Vdash^{v}$.

when it comes to formalized proofs the incentives and difficulties are reversed: you do have to provide complete proofs and the typechecker helps you keep track of whether you are missing something. So, after formalizing the fundamental lemma for the first logical relation, we abstracted over judgmental equality in the definition of the relation and collected the properties needed for the proof of the fundamental lemma, while also keeping an eye on what would be easily provable for algorithmic equality. Doing so, we obtained a notion we called "generic equality" which we use to parametrize the reducibility relation and its fundamental lemma[9]. We can then instantiate this parametrized result twice, for the two equalities of interest, to complete the overall proof of decidability of conversion.

While the article only handles a fairly minimal type theory, we believe this formalization can form a good basis for applying the proof technique to more complex theories. For example, the Kripke structure could be extended to contain more than just renamings, but also maps from the indexing category for type theories whose denotational model is based on presheaf models. An example of a similar setup can be seen in the proof of canonicity for Cubical Type Theory [23].

This formalization can also be seen as a step towards bootstrapping Agda, as it could reasonably be extended into a double checker for the core calculus.

1.1.4 Statement of Contribution

First part

- Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of Higher Truncations. In *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*

  Contributed the proof without extra higher inductive types and participated in writing the article.

Second Part

- Andreas Abel and Andrea Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *Programming Languages and Systems*, 2014

  Participated in the design and contributed most of the formalization of the strong normalization proof and collaborated to the writing of the article.

- Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*

  Developed GCTT and its prototype implementation in collaboration with Hans Bugge Grathwohl, contributed the proof that bisimulation is equivalent to equality, and participated in writing the article.

[9] I would like to insert here a special acknowlegment to Joakim, who performed the bulk of the formalization and derived this abstraction.

- Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017

  Contributed the application of parametricity to sizes and the proofs and write-up of the construction of (co)inductive types by induction on sizes and participated in the writing of other sections. Contributed to the formulation of the model as modified cubical sets and the use of the cohesive structure of the category. Contributed the use of a modified Glue construction to internalize naturality with regard to functions and the prototype implementation as a fork of Agda.

- Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1(ICFP):33:1–33:30, 2017

  Participated in the design of the type system and decidability of conversion proof.

Third Part

- Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *PACMPL*, 2(POPL):23:1–23:29, 2018

  Contributed to the formalization, in particular in setting up the logical relation and the reasoning with substitutions. Collaborated to the writing of the article.

Bibliography

[1] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *JFP*, 12(1):1–41, 2002.

[2] Andreas Abel, Thierry Coquand, and Bassel Mannaa. On decidability of conversion in type theory. 2016.

[3] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *PACMPL*, 2(POPL):23:1–23:29, 2018.

[4] Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *LMCS*, 8(1:29):1–36, 2012. TYPES'10 special issue.

[5] Andreas Abel and Andrea Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *Programming Languages and Systems*, 2014.

[6] Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1(ICFP):33:1–33:30, 2017.

[7] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM.

[8] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 503–515, New York, NY, USA, 2014. ACM.

[9] Robert Atkey, Patricia Johann, and Andrew Kennedy. Abstraction and invariance for algebraically indexed types. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, pages 87–100.

[10] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 197–208, New York, NY, USA, 2013. ACM.

[11] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[12] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 319:67 – 82, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[13] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*.

[14] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, Volume 8, Issue 4, October 2012.

[15] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 20–35, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[16] Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of Higher Truncations. In *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*.

[17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *TYPES'15*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[18] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. *A simple type-theoretic language: Mini-TT*, page 139–164. Cambridge University Press, 2009.

[19] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *LNCS*, pages 39–59. Springer, 1995.

[20] Georges Gonthier. Formal proof – the four colour theorem. *Notices of the AMS*, 55(11):1382–1392, 2008.

[21] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *ITP'13*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[22] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 209–220, New York, NY, USA, 2013. ACM.

[23] Simon Huber. Canonicity for cubical type theory. *CoRR*, abs/1607.04156, 2016.

[24] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423, 1996.

[25] Nicolai Kraus. Constructions with non-recursive higher inductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 595–604, New York, NY, USA, 2016. ACM.

[26] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92, 2001.

[27] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[28] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. *Electronic Notes in Theoretical Computer Science*, 276:263 – 289, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[29] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal universes in models of homotopy type theory. *CoRR*, abs/1801.07664, 2018.

[30] Per Martin-Löf. An intuitionistic theoy of types. In *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998. Reprinted version of an unpublished report from 1972.

[31] Conor Thomas McBride. How to keep your neighbours in order. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 297–309, New York, NY, USA, 2014. ACM.

[32] Hiroshi Nakano. A modality for recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, LICS '00, pages 255–, Washington, DC, USA, 2000. IEEE Computer Society.

[33] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers, Göteborg, Sweden, 2007.

[34] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017.

[35] Brigitte Pientka. Poplmark reloaded: Mechanizing logical relations proofs (invited talk). In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 1–1, New York, NY, USA, 2018. ACM.

[36] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[37] Emily Riehl and Micheal Shulman. A type theory for synthetic ∞-categories. *ArXiv e-prints*, 2017.

[38] Egbert Rijke and Bas Spitters. Sets in homotopy type theory. *Mathematical Structures in Computer Science*, 25(5):1172–1202, 2015.

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

[40] Andrea Vezzosi. Agda-flat, 2017. Branch of the Agda proof assistant, `https://github.com/agda/agda/tree/flat`.

[41] Andrea Vezzosi. Agda-parametric, 2017. Branch of the Agda proof assistant, `https://github.com/agda/agda/tree/cubical`.

[42] Andrea Vezzosi. Agda-parametric, 2017. Branch of the Agda proof assistant, `https://github.com/agda/agda/tree/parametric`.

[43] Andrea Vezzosi. Streams for cubical type thoery. `http://www.cse.chalmers.se/~vezzosi/streams-ctt.pdf`, 2017.

[44] David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. PhD thesis, Chalmers University of Technology, 2007.

Part I

Homotopy Type Theory

Chapter 2

Functions out of Higher Truncations

FUNCTIONS OUT OF HIGHER TRUNCATIONS

PAOLO CAPRIOTTI, NICOLAI KRAUS, AND ANDREA VEZZOSI

Abstract. In homotopy type theory, the truncation operator $\|-\|_n$ (for a number $n \geq -1$) is often useful if one does not care about the higher structure of a type and wants to avoid coherence problems. However, its elimination principle only allows to eliminate into n-types, which makes it hard to construct functions $\|A\|_n \to B$ if B is not an n-type. This makes it desirable to derive more powerful elimination theorems. We show a first general result: If B is an $(n+1)$-type, then functions $\|A\|_n \to B$ correspond exactly to functions $A \to B$ which are constant on all $(n+1)$-st loop spaces. We give one "elementary" proof and one proof that uses a higher inductive type, both of which require some effort. As a sample application of our result, we show that we can construct "set-based" representations of 1-types, as long as they have "braided" loop spaces. The main result with one of its proofs and the application have been formalised in Agda.

1. Introduction

As it is very well-known, the type constructor Σ of Martin-Löf type theory expresses a very strong form of existence. Although a type of the form $\Sigma\,(a : A)\,.\,P(a)$ is read as "there exists an element in A for which the predicate P holds" under the *propositions as types* view, an element of such a type is more than a proof of mere existence: it includes a very concrete example of an element $a : A$. This is not always satisfying as, for example, the set-theoretic axiom of choice becomes a tautology when translated naively to type theory. The idea of adding a construction which allows to formulate existence in a weaker sense has been studied intensively in various different settings. As far as we know, the first documented appearance are *squash types* in the extensional theory of NuPRL [7]. Later, Awodey and Bauer introduced a similar concept in extensional Martin-Löf type theory, called *bracket types* [4]. Homotopy type theory has introduced the *propositional truncation* operation, written $\|-\|_{-1}$ or simply $\|-\|$ [14]. It forces all elements to be equal, in the sense that the identity type $x = y$ is inhabited for any $x, y : \|A\|_{-1}$, and it is well-known that $x = y$ will in fact be uniquely inhabited (i.e. equivalent, or isomorphic, to the unit type). Classically, $\|A\|_{-1}$ is always equivalent to either the unit type or the empty type, but this is of course not the case in a constructive setting.

The homotopical view has suggested that propositional truncation is only one out of infinitely many operations that reduce the complexity of a type. As "types are weak ω-groupoids" ([12] and [15]), it is easy to imagine that there is, for every number $n \geq -1$, an operation which trivialises all the structure above level $(n+1)$. In other words, this is a reflector for the category of weak n-groupoids, viewed as a subcategory of weak ω-groupoids, roughly speaking. In homotopy type theory, we write this operation as $\|-\|_n$ ("n-truncation"), and it can be seen and implemented as a *higher inductive type* [14]. The truncation operator $\|-\|_n$ is a *monad* in some

1991 *Mathematics Subject Classification.* F.4.1 Mathematical Logic.

Key words and phrases. homotopy type theory, truncation elimination, constancy on loop spaces.

Nicolai Kraus acknowledges support by the Engineering and Physical Sciences Research Council (EPSRC), grant reference EP/M016994/1.

appropriate sense (and even a *modality* in the sense of [14]), and if we want to, we can choose to work completely in that monad. Types that are canonically equivalent to their n-truncation are called *n-types*, or *n-truncated types*.

Considering n-types (for some given n) instead of all types is useful if we do not care about or want to avoid potential higher equality proofs. For example, if we formalise algebraic structures such as groups, we may require that the type of group elements is of truncation level 0 in order to match the set-theoretic definition: equality of group elements should be a mere proposition and not carry additional information, that is, there is at most one proof that given group elements are equal. As a consequence, for any type A with an element $a : A$, the type $a = a$ is not necessarily a group. It does have a neutral element an elements can be inverted and composed, corresponding to the fact that equality is reflexive, symmetric, and transitive. However, $a = a$ is not a 0-truncated type. We can use 0-truncation to make up for this, and $\|a = a\|_0$ is indeed a group, called the *fundamental group* of A at basepoint a, while $a = a$ (as *pointed type* also written $\Omega(A, a)$) is the *loop space* at point a.

A drawback of truncations is that it can be hard to get out of them, that is, "to leave the monad". A priori we have, for any type A and number $n \geq -1$, a map $|-| : A \to \|A\|_n$, but there is in general no function in the other direction. The universal property of $\|-\|_n$ says that, via composition with $|-|$, the type of functions $\|A\|_n \to B$ is equivalent to the type $A \to B$, but only if B is n-truncated. To continue with the previous example, an element of the fundamental group of A at basepoint a is really an equivalence class of equality proofs (or *paths*) between a and itself, but it is in general impossible to get a specific representative from such a class; that is, we cannot construct a section of the map $|-| : (a = a) \to \|a = a\|_0$. Of course, we would not have expected anything else: it is unreasonable to assume that we can make this sort of choice without any further assumptions. Although the truncation operator $\|-\|_n$ is often described as "cutting of" higher structure of a type, it is more accurate to think of it as "filling non-trivial loops", which makes it plausible that it is *harder* instead of *easier* to define a function out of $\|A\|_n$ than out of A.

Unlike in the example above, it is in some cases reasonable to expect that we can get a function $\|A\|_n \to B$ even if B is not an n-type. If $\|A\|_{-1}$ tells us that A has some element without revealing a concrete one to us, then a function $\|A\|_{-1} \to B$ should be the same as a function $f : A \to B$ which cannot look at the "input".[1] What exactly this means is difficult to state in general (see [8]), so let us restrict ourselves to the case that B is 0-truncated (also called a *set*). In this case, the statement that "f does not look at its input" can be expressed by saying that f maps any pair of inputs to equal values, $\Pi_{x,y:A}(f(x) = f(y))$. Indeed, it has been shown that a function f with this behaviour gives rise to a map $\|A\|_{-1} \to B$ [10].

Even if we have a function $A \to B$, it can be very hard to tell whether it is possible to construct a function $\|A\|_n \to B$ unless B is an n-type, and if it is possible, there is no direct way to do so as the universal property (or the elimination principle) cannot be applied directly. The usual workaround is looking for an n-type C "in the middle", that is such that one has functions $A \to C$ and $C \to B$. One can then apply the elimination principle to construct a function $\|A\|_n \to C$ which, by composition, yields a function $\|A\|_n \to B$ as desired. The type C is constructed ad-hoc, and it is natural to ask for a more powerful elimination principle (or universal property)

[1] This only makes sense if stated internally. Of course, a concrete implementation of f *can* compute differently if applied to different terms of type A. As long as we stay inside the theory, we cannot talk about judgmental equality.

of $\|-\|_n$ which allows the construction of functions $\|A\|_n \to B$ in a more principled and streamlined way.

This has been done for the (-1)-truncation in previous work [8], where it is shown that functions $\|A\|_{-1} \to B$ correspond exactly to functions $A \to B$ with an infinite tower of coherence conditions. This can be understood as a generalised version of the usual universal property of $\|-\|_{-1}$. If B is known to be n-truncated for some fixed finite n, the infinite tower becomes finite and can be expressed directly in type theory, whereas the existence of *Reedy limits* [13] is necessary for the general case. If B is a 0-type, the "tower" of coherence condition is exactly the single condition $\Pi_{x,y:A}(f(x) = f(y))$ discussed above. If B is even a (-1)-type itself, the tower vanishes completely and the usual universal property remains. Unfortunately, it seems that there is no immediate generalisation of the proof of [8] to n-truncations.

In this paper, we do consider n-truncations for general n, but we assume that B is $(n+1)$-truncated, and already this case seems to be involved. We show that functions $\|A\|_n \to B$ correspond exactly to those functions $A \to B$ that are constant on all $(n+1)$-st loop spaces. We offer two proofs for this fact, one which works in "plain" homotopy type theory with general truncations, and the other involving a higher inductive type. The first proof, which we call the "elementary proof", is close to not even requiring the univalence axiom (the central concept of homotopy type theory expressing that equality in the universe is given by type equivalence). The only reason why univalence is necessary is that we need to be able to translate between truncations ($\|a =_A b\|_n$ is equivalent to $|a| =_{\|A\|_{n+1}} |b|$). The second proof (Section 4) uses an argument that makes crucial use of both a higher inductive type and the univalence axiom, and we therefore call it the "HIT proof". In the HIT proof, we will construct a higher inductive type in such a way that it is the "initial" type through which functions $f : A \to B$ with the property (10) factor, and we will show that this type *is* really $\|A\|_n$. Although we show an equivalence of types, we believe that the main application is the construction of functions $\|A\|_n \to B$, that is, one may often want to use only one direction of the equivalence. Therefore, the result can be used as an elimination principle that is more powerful than the usual recursion principle of the truncation. We also present a sample application (a translation of types into "set-based representation"), and conclude with a discussion on how the generalised statement should look like, and under which assumptions it should be provable.

The main contents of this paper have, in slightly different form, appeared in the second-named author's Ph.D. thesis [9].

Outline. We start by stating the result of the paper in Section 2, and discuss two special cases ($n \equiv -1$ and $n \equiv 0$). In Section 3, we give the "elementary" proof of this result, and in Section 4, the (technically harder, but conceptually clear) proof that uses a higher inductive type. We discuss a sample application of the case $n \equiv 0$ in Section 5, namely a construction of a *set-based representation* of any given type, provided that it fulfils a property that e.g. loop spaces do. Finally, in Section 6, we compare the two proofs with each other. We also compare our result with the *general universal property of the propositional truncation* as proved before [8], and discuss why the potential generalisations seem so much more involved than what we have done here.

Setting. We consider the theory of the standard reference on homotopy type theory, that is, the textbook [14]. To summarise, we need a version of intensional Martin-Löf type theory with Σ, Π, and identity types. In addition, we assume that the theory has a univalent universe, and that there are truncation operators $\|-\|_n$ for all $n \geq -1$, with the canonical projections $|-| : A \to \|A\|_n$. This concept is explained in detail in [14, Chap. 7.3]). The statement and the first proof that we

give do not need higher inductive types [14, Chap. 6] other than the truncations, while the second proof that we give makes heavy use of such a higher inductive type.

Agda Formalisation. We have formalised the main result, together with the "elementary" proof (Section 3) and the sample application (Section 5), in Agda [6]. The source code can be found on GitHub, at `github.com/pcapriotti/agda-base/tree/trunc`. The results of this paper are contained in the module `hott.truncation.elim`. A browsable HTML version of the formalisation can be accessed at `paolocapriotti.com/agda-base/trunc/hott/truncation/elim.html`. We encourage a reader who is not familiar with Agda to have a look at the latter, which does not need any software apart from a web browser. For all the technical details, we refer to the readme file in the repository.

On a minor note, we have chosen not to make use of the common (but, as far as we know, not justified by a formal argument) hack that makes truncations satisfy the judgmental computation rule. As we wanted our formalisation to be readable, this has required us to think of some implementation strategies that make the code in this setting more elegant than the "straightforward" formalisation approaches.

2. The Statement of the Theorem

Let us begin by clarifying some notation. In general, we stick closely to the terminology of the standard reference on the topic, the textbook [14]. We write $\Pi_{a:A} B(a)$ for Π-types as it is done there, but $\Sigma\,(a : A)\,.\,B(a)$ for Σ-types.[2] For better readability, we uncurry implicitly and write $f(a, b) : C$, even if f is a function of type $A \to B \to C$. Instead of $\lambda h.h \circ g$, we write $\_ \circ g$. By the *distributivity law of* Σ *and* Π, we mean the well-known equivalence

$$\Pi_{a:A}\Sigma\,(b : B(a))\,.\,C(a, b) \;\simeq\; \Sigma\,(g : \Pi_{a:A} B(a))\,.\,\Pi_{a:A} C(a, g(a)), \tag{1}$$

sometimes called the *type-theoretic axiom of choice*. As it is standard [14], we write $\mathsf{is}\text{-}n\text{-}\mathsf{type}(A)$ for the propositional type expressing that A is n-truncated if $n \geq -2$ is an integer, defined by

$$\mathsf{is}\text{-}(-2)\text{-}\mathsf{type}(A) :\equiv \Sigma\,(a_0 : A)\,.\,\Pi_{a:A} a = a_0 \tag{2}$$

$$\mathsf{is}\text{-}(n+1)\text{-}\mathsf{type}(A) :\equiv \Pi_{a_1,a_2:A} \mathsf{is}\text{-}n\text{-}\mathsf{type}(a_1 = a_2), \tag{3}$$

and the special case when n is -2 ("A is contractible") is also written as $\mathsf{isContr}(A)$. We assume that there is a universe $\mathcal{U}$, and we write $\mathcal{U}^n$ for the type (or "universe") of n-types in $\mathcal{U}$ (cf. [14, Chap. 7.1]),

$$\mathcal{U}^n :\equiv \Sigma\,(X : \mathcal{U})\,.\,\mathsf{is}\text{-}n\text{-}\mathsf{type}(X). \tag{4}$$

Further, we write $\mathcal{U}_\bullet$ for the type (or "universe") of pointed types in $\mathcal{U}$ (cf. [14, Def. 2.1.7]),

$$\mathcal{U}_\bullet :\equiv \Sigma\,(X : \mathcal{U})\,.\,X. \tag{5}$$

If we have a type A and a pointed type (B, b), together with a function $f : A \to B$, we say that "f is null" if it is constantly b, that is,

$$\mathsf{isNull}(f) :\equiv \Pi_{x:A} b = f(x). \tag{6}$$

Recall that there is an endofunction on $\mathcal{U}_\bullet$, the *loop space function* Ω,

$$\Omega(A, a) :\equiv (a = a, \mathsf{refl}_a)\,. \tag{7}$$

For any natural number n, we can iterate this endofunction n times, for which we write Ω^n. Instead of $\pi_1\,(\Omega^n(A, a))$ and $\pi_1((\Omega(A, a)))$, we simply write $\Omega^n_t(A, a)$

[2] This seemingly inconsistent notation is intentional: we sometimes have nested Σ-types, e.g. $\Sigma\,(a : A)\,.\,\Sigma\,(b : B(a))\,.\,C(a, b)$, and we view the components as "equally valued"; thus, writing exactly one component bigger than the others would not look correct.

and $\Omega_t(A,a)$ if we want to talk about the underlying type (i.e. ignore the point). Further, given two types A and B together with any function $f : A \to B$ and a point $a : A$, we have a function

$$\mathsf{ap}_{f,a} : \Omega_t(A,a) \to \Omega_t(B, f(a)). \tag{8}$$

In the same way, we have (given A, B, f as before) $\mathsf{ap}^n_{f,a} : \Omega^n_t(A,a) \to \Omega^n_t(B, f(a))$, and Ω is really an endofunctor in some appropriate sense.[3]

Our result can now be stated as follows:

Theorem 2.1. *Let $n \geq -1$ be a number, A a type, and B an $(n+1)$-type. Assume that $f : A \to B$ is a function. Then, f can be factored through the n-truncation, that is*

$$\Sigma\,(f' : \|A\|_n \to B)\,.\, f' \circ |-| = f, \tag{9}$$

if and only if $\mathsf{ap}^{n+1}_{f,a}$ is null for every a,

$$\Pi_{a:A} \mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a}), \tag{10}$$

and both of the types (9) *and* (10) *are propositional.*

An immediate corollary tells us how we can eliminate out of truncations:

Corollary 2.2. *Assume we have n, A and B as in Theorem 2.1. If we want to construct a function $\|A\|_n \to B$, it suffices to find a function $f : A \to B$ which satisfies $\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a})$.*

Before approaching a proof of Theorem 2.1, let us have a look at two special cases, namely the cases $n \equiv -1$ and $n \equiv 0$. The first case is known [10] and will serve as the base case for the two general proofs presented later. The second case is not strictly necessary, but serves to exemplify the techniques used in the "HIT proof" (Section 4).

The case $\mathbf{n \equiv -1}$: The simplified statement of Theorem 2.1 reads in this case as follows: Assume we are given a type A and a 0-type B (often called a *set*). A function $f : A \to B$ factors through the propositional truncation if and only if

$$\Pi_{x,y:A} f(x) = f(y). \tag{11}$$

This follows easily from previous work, e.g. [8, Prop. 2.2]. It is a pleasant surprise that "$\mathsf{ap}^0_{f,a}$ is null for all a", simply by unfolding our definitions, simplifies to (11), which is "f is constant" in the sense of [10].[4]

The case $\mathbf{n \equiv 0}$. Here, our result (Theorem 2.1) implies that, for any type A and 1-type B, a function $f : A \to B$ factors through $\|A\|_0$ if and only if, for all $a : A$ and $p : a = a$, we have that $\mathsf{ap}_{f,a}(p)$ equals $\mathsf{refl}_{f(a)}$. As Shulman has remarked in an online discussion (in the comment section of a blog post [5]), this follows from the *Rezk completion* [1]: Let $\tilde{A}$ be the precategory with the type A of objects and $\mathsf{hom}(a_1, a_2) :\equiv \|a_1 =_A a_2\|_{-1}$, and let $\tilde{B}$ be the category with B as objects and $\mathsf{hom}(b_1, b_2) :\equiv (b_1 =_B b_2)$. Then, f with the condition $\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}_{f,a})$ gives (already using the case $n \equiv -1$) rise to a functor $\tilde{A} \to \tilde{B}$. Such a functor generates a functor between the Rezk completion of $\tilde{A}$ and the category $\tilde{B}$, and the former happens to be $\|A\|_0$.

In the remainder of the current section, we give a simple technical construction which essentially serves as a reformulation of Theorem 2.1 and which is necessary for both the elementary and the HIT proof. For types A and B, assume we are

[3] Of course, $\mathsf{ap}_{f,a}$ is its action on the morphism f and could thus rightfully be called $\Omega(f,a)$.

[4] In the simplified formulation, we have omitted the part that the two logically equivalent types are propositional. This is easy to see here, and will in the general case be part of the proof.

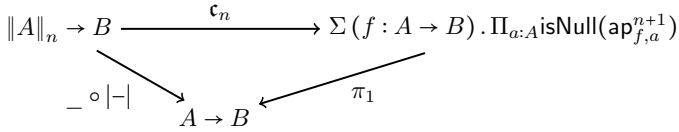

FIGURE 1. The canonical map $\mathfrak{c}_n$ as map between fibres

given a function $g : \|A\|_n \to B$. We can consider the composition $A \xrightarrow{|-|} \|A\|_n \xrightarrow{g} B$. For any $a : A$ we have, by functoriality of Ω^{n+1}, that the composition

$$\Omega^{n+1}_t(A, a) \xrightarrow{\mathsf{ap}^{n+1}_{|-|,a}} \Omega^{n+1}_t(\|A\|_n, |a|) \xrightarrow{\mathsf{ap}^{n+1}_{g,|a|}} \Omega^{n+1}_t(B, g(|a|)) \tag{12}$$

is equal to $\mathsf{ap}^{n+1}_{g\circ|-|,a}$. But $\Omega^{n+1}_t(\|A\|_n, |a|)$ is contractible ([14, Thm. 7.2.9]), and $\mathsf{ap}^{n+1}_{g,|a|}$ clearly maps its unique element to the basepoint of $\Omega^{n+1}(B, g(|a|))$. Therefore, $\mathsf{ap}^{n+1}_{g\circ|-|,a}$ is null. From this construction, we get a canonical function

$$\mathfrak{c}_n : (\|A\|_n \to B) \to \Sigma\,(f : A \to B)\,.\,\left(\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a})\right). \tag{13}$$

We then claim the following:

Lemma 2.3 ("Total space" formulation of Theorem 2.1)**.** *For any $n \geq -1$, any type A and any $(n+1)$-type B, the types $\|A\|_n \to B$ and $\Sigma\,(f : A \to B)\,.\,\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a})$ are equivalent, and the equivalence is given by the canonical function $\mathfrak{c}_n$.*

It is easy to see that Lemma 2.3 does indeed imply, and is nearly immediately equivalent to, Theorem 2.1. Consider the triangle shown in Figure 1, where the top horizontal map is the canonical map $\mathfrak{c}_n$, the left one is composition with $|-|$, and the right one is simply the projection. The triangle clearly commutes (judgmentally) by construction. Let us fix some function $f : A \to B$. The fibre (or "inverse image") over f is, in the case of $\_ \circ |-|$, exactly (9), i.e. the statement that f can be lifted. In the second case, the fibre is (10). Therefore, $\mathfrak{c}_n$ induces an equivalence of the two fibres, which implies that $\mathfrak{c}_n$ itself is an equivalence (see [14, Thm. 4.7.7]).

3. The "Elementary" Proof

In this section, we give our first proof of Lemma 2.3 (and thereby of Theorem 2.1). This does not need higher inductive types apart from truncations that already appear in the statement. The idea is to not prove the result for *any* type A first, but only for an n-connected one.[5] Afterwards, we generalise this to arbitrary types, by splitting the type into its "connected components" and gluing together the constructions for the components.

Lemma 3.1. *If $n \geq -1$ be a number, A an n-connected type, and B be an $(n+1)$-type, the canonical map $\mathfrak{c}_n$ is an equivalence.*

Proof. We do induction on n. As already discussed above, the case that n is -1 is known (e.g. [8, Prop. 2.2]).

Let now $n \geq 0$ be any given number. Note that, due to the assumption that $\|A\|_n$ is contractible, we have a unique element $x_0 : \|A\|_n$, the type $\|A\|_n \to B$ is actually equivalent to B, and any function $g : \|A\|_n \to B$ is uniquely specified by its value $g(x_0)$.

[5] Recall that a type A is n-connected if $\|A\|_n$ is contractible [14, Def. 7.5.1].

The claim of the lemma is propositional. Applying the eliminator of $\|A\|_n$, we may not only assume that we are given $x_0 : \|A\|_n$, but we can also assume a point $a : A$. A potential inverse of $\mathfrak{c}_n$ is then given by[6]

$$\mathfrak{d}_n : \left(\Sigma\,(f : A \to B)\,.\,\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a})\right) \to (\|A\|_n \to B) \tag{14}$$

$$\mathfrak{d}_n(f,p) :\equiv \lambda\_.f(a). \tag{15}$$

To show that $\mathfrak{c}_n$ and $\mathfrak{d}_n$ are inverses, we check that both compositions are the identities. One direction is easy: for any $g : \|A\|_n \to B$, we have

$$\mathfrak{d}_n(\mathfrak{c}_n(g))(x_0) \equiv g(|a|), \tag{16}$$

and the latter is equal to $g(x_0)$.

For the other direction, assume we have $f : A \to B$ together with a proof q. We need to show $(f,q) = \mathfrak{c}_n(\mathfrak{d}_n(f,q))$. Fortunately, the equality of the two second components is automatic thanks to the fact that $\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a})$ is propositional, and we only need to prove the equality of f and $\pi_1(\mathfrak{c}_n(\mathfrak{d}_n(f,q)))$. We observe that the latter expression computes to $\lambda\_.f(a)$. Thus, our goal is to show that, for any $a' : A$, we have $f(a) = f(a')$.

We use the induction hypothesis with $(a = a')$ for A, and $f(a) = f(a')$ for B. By the connectedness assumption on A, the type $|a| = |a'|$ is contractible. Consequently, the type $\|a = a'\|_{n-1}$ is contractible ([14, Thm. 7.3.12], note that this theorem depends on the univalence axiom). Put differently, $(a = a')$ is $(n-1)$-connected. As B is an $(n+1)$-type, we know that $f(a) = f(a')$ is n-truncated. By the induction hypothesis, it is hence enough to construct an element of

$$\Sigma\,(k : a = a' \to f(a) = f(a'))\,.\,\Pi_{p:a=a'}\mathsf{isNull}(\mathsf{ap}^n_{k,p}). \tag{17}$$

For k, we choose ap_f. By path induction, we may assume that p is refl_a. Thus, we need to show that $\mathsf{ap}^n_{\mathsf{ap}_{f,a},\mathsf{refl}_a}$ is null. This term is equal to $\mathsf{ap}^{n+1}_{f,a}$.[7] The condition that this function null is exactly what is given by $q(a')$. □

To move from n-connected to arbitrary types A, we simply split a type into n-connected components. This is very intuitive for $n \equiv 0$, in which case we use that any type (or "space") can be viewed as the "disjoint sum" of its connected components. To be precise, an element of a component is a point of A together with a proof that it is in the component. For $n \equiv 0$, this proof is propositional. For higher n, it is not. This makes the general case less intuitive and hard to picture. In fact, the proof determines in which component the element is, which makes it seem circular. Fortunately, it is easier to write down the type-theoretic argument than picturing the topological intuition, as we will see in the following lemma.

Lemma 3.2. *For any type A and number n, we define the family of n-connected components,*

$$\mathsf{conn}_n : \|A\|_n \to \mathcal{U} \tag{18}$$

$$\mathsf{conn}_n(x) :\equiv \Sigma\,(a : A)\,.\,x =_{\|A\|_n} |a|. \tag{19}$$

Then, for any $x : \|A\|_n$, the type $\mathsf{conn}_n(x)$ is n-connected. Further, "choosing an n-connected component and then a point in this component" corresponds to "choosing a point", that is,

$$\Sigma\,(x : \|A\|_n)\,.\,\mathsf{conn}_n(x) \;\simeq\; A. \tag{20}$$

[6] We use _ if we do not need to give the bound variable a name.

[7] Depending on the the exact definition of ap^n, this can hold judgmentally, but can also be rather involved. We refer to our formalisation for technical details.

Proof. This is easy and standard. For the first part, we claim that the equivalence

$$\left\|\Sigma\,(a : A)\,.\,x =_{\|A\|_n} |a|\right\|_n \;\simeq\; \Sigma\,(y : \|A\|_n)\,.\,x =_{\|A\|_n} y \tag{21}$$

holds, where the left-hand type is $\|\mathsf{conn}_n(x)\|_n$ by definition, and the right-hand type has the form of a *singleton*.[8] For both directions of (21), we apply the dependent eliminator of $\|-\|_n$. From left to right, we map $|(a,p)|$ to $(|a|,p)$. From right to left, we map $(|a|,p)$ to $(|a,p|)$. For an alternative proof, see [14, Cor. 7.5.8].

To see that the equivalence (20) holds, it is enough to unfold the definition of conn_n, and use that in $\Sigma\,(x : \|A\|_n)\,.\,\Sigma\,(a : A)\,.\,x =_{\|A\|_n} |a|$, the first and the third component form a singleton. □

Finally, we can complete the first proof of our main result:

"Elementary" proof of Lemma 2.3. Assume we have n, A, and B as in the statement. The preceding two lemmata tell us that, for any $x : \|A\|_n$, the canonical map

$$\mathfrak{c}_n^x : B \to \left(\Sigma\,(f_x : \mathsf{conn}_n(x) \to B)\,.\,\Pi_{y:\mathsf{conn}_n(x)}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f_x,y})\right) \tag{22}$$

is an equivalence (note that we have omitted the contractible type $\|\mathsf{conn}_n(x)\|_n$ in the domain of $\mathfrak{c}_n^x$). A family of equivalences gives rise to an equivalence of families, so that we get that the map

$$\tilde{\mathfrak{c}}_n : (\|A\|_n \to B) \;\to\; \left(\Pi_{x:\|A\|_n}\Sigma\,(g_x : \mathsf{conn}_n(x) \to B)\,.\,\Pi_{y:\mathsf{conn}_n(x)}\mathsf{isNull}(\mathsf{ap}^{n+1}_{g,y})\right) \tag{23}$$

$$\tilde{\mathfrak{c}}_n(k) :\equiv \lambda x.\mathfrak{c}_n^x(k(x)) \tag{24}$$

is also an equivalence.

All we need at this point is an equivalence from the codomain of the function (24) to the type stated in the theorem, i.e. $\Sigma\,(f : A \to B)\,.\,\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,a})$, and the composition of (24) and this equivalence has to be the canonical map $\mathfrak{c}_n$. We calculate:

$$\Pi_{x:\|A\|_n}\Sigma\,(g_x : \mathsf{conn}_n(x) \to B)\,.\,\Pi_{y:\mathsf{conn}_n(x)}\mathsf{isNull}(\mathsf{ap}^{n+1}_{g_x,y}) \tag{25}$$

(by the distributivity law)

$$\simeq\quad \Sigma\left(g : \Pi_{x:\|A\|_n}(\mathsf{conn}_n(x) \to B)\right)\,.\,\Pi_{x:\|A\|_n}\Pi_{y:\mathsf{conn}_n(x)}\mathsf{isNull}(\mathsf{ap}^{n+1}_{g(x),y}) \tag{26}$$

(by currying and using the canonical equivalence (20))

$$\simeq\quad \Sigma\,(h : A \to B)\,.\,\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{\lambda y:\mathsf{conn}_n(|a|).h(\pi_1 y),(a,\mathsf{refl}_{|a|})}) \tag{27}$$

Fortunately, the (pointed) types $\Omega^{n+1}(\mathsf{conn}_n(|a|),(a,\mathsf{refl}_{|a|}))$ and $\Omega^{n+1}(A,a)$ are equivalent, with the equivalence being $\mathsf{ap}^{n+1}_{\pi_1}$; this is an easy technical statement that follows from [11, Lem. 5.1]. If we compose $\mathsf{ap}^{n+1}_{\lambda y:\mathsf{conn}_n(|a|).h(\pi_1 y),(a,\mathsf{refl}_{|a|})}$ with the inverse of this equivalence, functoriality of ap^{n+1} allows us to simplify the expression.

$$\simeq\quad \Sigma\,(h : A \to B)\,.\,\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{h,a}) \tag{28}$$

We need to check that the composition of $\tilde{\mathfrak{c}}_n$ with this equivalence is indeed the canonical function $\mathfrak{c}_n$. This is immediate as we only need to check that the first component (the map $A \to B$) turns out to be the correct function, as the second component is propositional. □

[8] If $z_0 : Z$ is some point of some type, we call any type of the form $\Sigma\,(z : Z)\,.\,z = z_0$ a *singleton*. It is well-known that singletons are contractible and therefore "neutral" components of Σ-types, which we use here and later.

4. The "HIT Proof"

Our second proof is fairly technical. We construct a higher inductive type with a suitable elimination property and show that it is equivalent to $\|A\|_n$. As a preparation, we show a small lemma. It is a part of a theorem that has been introduced in [9], where it is described as *local generalised Hedberg argument*.

Lemma 4.1 (main part of [9, Thm. 3.2.1])**.** *Let (A, a_0) be a pointed type. Assume further that P is a* pointed *family of $(n-1)$-types over (A, a_0), that is, a family $P : A \to \mathcal{U}^{n-1}$ with a point $p_0 : P(a_0)$. If $P(a)$ implies that a_0 is equal to a, i.e. $m : \Pi_{a:A} P(a) \to a_0 = a$, then A is "locally an n-type" in the sense that $\Omega^{n+1}(A, a_0)$ is contractible.*[9]

Proof sketch. Consider the following composition of three maps, for any $a : A$:

$$a_0 = a \xrightarrow{q \mapsto \mathsf{transport}^P(q, p_0)} P(a) \xrightarrow{m_a} a_0 = a \xrightarrow{q \mapsto m_{a_0}(p_0) \cdot q} a_0 = a$$

By path induction, we easily see that these maps make $a_0 = a$ a retract of $P(a)$. Hence, the former is $(n-1)$-truncated [14, Thm. 7.1.4], which shows the claim [14, Thm. 7.2.9]. □

We are ready to define the higher inductive type that plays the central role in the second proof of Lemma 2.3. For the following definition and for the rest of the section, we fix a type A and a number $n \geq -1$.

Definition 4.2. Define the higher inductive type H, which depends on A and n, as given by the constructors

$$\eta : A \to H \tag{29}$$

$$\epsilon : \Pi_{a,b:A} \left(\|a = b\|_{n-1} \to \eta(a) = \eta(b) \right) \tag{30}$$

$$\delta : \Pi_{a:A} \left(\mathsf{refl}_{\eta(a)} =_{\eta(a)=\eta(a)} \epsilon(a, a, |\mathsf{refl}_a|) \right) \tag{31}$$

$$t : \mathsf{is\text{-}}(n+1)\text{-}\mathsf{type}(H). \tag{32}$$

The complicated looking constructors ϵ and δ are more intuitive than they looks at first sight. If we have $(a = b)$, we of course always get a proof of $\eta(a) = \eta(b)$ using ap_η. The constructor ϵ says that $\|a = b\|_{n-1}$ is sufficient, while δ ensures that ϵ is really a lifting of ap_η through $\|a = b\|_{n-1}$. This is because we could have used the expanded form

$$\delta' : \Pi_{a,b:A} \Pi_{p:a=b} \left(\mathsf{ap}_\eta(p) =_{\eta(a)=\eta(b)} \epsilon(a, b, |p|) \right), \tag{33}$$

instead of the constructor δ. By path induction on p, the type (33) is easily seen to be equivalent to the original type (31). While (33) might look more regular next to (30), we choose (31) just for simplicity.

The recursion principle for H is straightforward to write down. Given some $(n+1)$-type B, we need a function $f : A \to B$, together with a function

$$k : \Pi_{a,b:A} (\|a = b\|_{n-1}) \to f(a) = f(b) \tag{34}$$

and a proof

$$h : \Pi_{a:A} \mathsf{refl}_{f(a)} =_{f(a)=f(a)} k(a, a, |\mathsf{refl}_{f(a)}|), \tag{35}$$

[9] This "local" form directly implies the "global" form: We can consider a relation $R : A \times A \to \mathcal{U}^{n-1}$ which implies identity and which has points $r_a : R(a, a)$ for all $a : A$; then, the lemma shows that A is an n-type.

we get a function $H \to B$ with the expected properties. It is more involved, nevertheless not inherently difficult, to state the induction principle following the standard ("intuitive") approach as used in [14, Chap. 6]. Given an $(n+1)$-truncated family $P : H \to \mathcal{U}^{n+1}$, in order to prove $\Pi_{x:H} P(x)$, we need

$$\overline{\eta} : \Pi_{a:A} P(\eta(a)) \tag{36}$$

$$\overline{\epsilon} : \Pi_{a,b:A} \Pi_{q:\|a=b\|_{n-1}} \mathsf{transport}^P(\epsilon(a,b,q), \overline{\eta}(a)) =_{P(\eta(b))} \overline{\eta}(b) \tag{37}$$

$$\overline{\delta} : \Pi_{a:A} \left(\mathsf{transport}^{\lambda r.\mathsf{transport}^P(r,\overline{\eta}(a))=\overline{\eta}(a)} \left(\delta(a), \mathsf{refl}_{\overline{\eta}(a)}\right) = \overline{\epsilon}(a,a,|\mathsf{refl}_a|) \right). \tag{38}$$

The above type expressions look rather involved. Fortunately, we do not need to deal too much with them at all because we are only interested in the case that P is n-truncated (instead of, more generally, $(n+1)$-truncated), which enables us to use the following observation:

Lemma 4.3 (Restricted dep. universal property of H)**.** *Given A and $n \geq -1$ as above and a family of n-types, $P : H \to \mathcal{U}^n$, the canonical map*

$$\Pi_{x:H} P(x) \xrightarrow{-\circ\eta} \Pi_{a:A} P(\eta(a)) \tag{39}$$

is an equivalence.

Proof. As P is a family of n-types, the type $\mathsf{transport}^P(\epsilon(a,b,q), \overline{\eta}(a)) =_{P(\eta(b))} \overline{\eta}(b)$, appearing in (37) as the target of $\overline{\epsilon}$, is $(n-1)$-truncated. By the standard universal property of the $(n-1)$-truncation, we may thus assume that the q in the type (37) is of the form $|p|$ with $p : a = b$, and then do path induction on p. This shows that the type of $\overline{\epsilon}$ is equivalent to

$$\overline{\epsilon}'' : \Pi_{a:A} \mathsf{transport}^P(\epsilon(a,a,|\mathsf{refl}_a|), \overline{\eta}(a)) =_{P(\eta(a))} \overline{\eta}(a). \tag{40}$$

Under this equivalence, the type of $\overline{\delta}$ becomes

$$\overline{\delta}'' : \Pi_{a:A} \left(\mathsf{transport}^{\lambda r.\mathsf{transport}^P(r,\overline{\eta}(a))=\overline{\eta}(a)} \left(\delta(a), \mathsf{refl}_{\overline{\eta}(a)}\right) = \overline{\epsilon}''(a) \right). \tag{41}$$

We see that the dependent pair of (40) and (41) forms a family of singletons. Therefore, there is always a canonical and unique choice for $\overline{\epsilon}$ and $\overline{\delta}$. The induction principle can therefore be simplified to only (36). Let us write $\mathsf{rind} : \Pi_{a:A} P(\eta(a)) \to \Pi_{x:H} P(x)$ for this *restricted induction principle*. It is easy to check that rind is indeed an inverse of the map $\_ \circ \eta$:

- For any $f : \Pi_{a:A} P(\eta(a))$ and $a : A$, the expression $(\mathsf{rind}(f) \circ \eta)(a)$ can be reduced to $f(a)$.
- For any $g : \Pi_{x:H} P(x)$, assume $x : H$. We need to show $(\mathsf{rind}(g \circ \eta))(x) = g(x)$. Using the restricted induction principle, we may assume $x \equiv \eta(a)$, and the left side can be reduced to the right side of the equation. □

This allows us to conclude the following crucial property of H:

Lemma 4.4. *The type H is n-truncated.*

Proof. It suffices to show that $\Omega^{n+1}(H,x)$ is contractible for all $x : H$ [14, Lem. 7.2.9]. The restricted induction principle of H tells us that, in order to show $P(x) :\equiv \mathsf{isContr}\left(\Omega^{n+1}(H,x)\right)$ for all x, we only need to prove $P(\eta(a_0))$ for any $a_0 : A$. Let us define a type family $Q : H \to \mathcal{U}^{n-1}$ using the restricted induction principle, $Q(\eta(a)) :\equiv \|a_0 = a\|_{n-1}$. This family is trivially inhabited at a_0. We want to show that Q implies local equality in the sense of $\Pi_{x:H}(Q(x) \to \eta(a_0) = x)$, and as this type family is n-truncated, we apply the restricted induction principle again and the goal becomes

$$\Pi_{a:A}(Q(\eta(a)) \to \eta(a_0) = \eta(a)). \tag{42}$$

By definition of Q, this is exactly given by the constructor ϵ, applied on a_0 and a.

This allows us to conclude, by Lemma 4.1, that H is n-truncated, as claimed. □

It is straightforward and standard that an n-truncated type which satisfies the dependent eliminating principle of $\|A\|_n$ is necessarily equivalent to $\|A\|_n$, and we record:

Corollary 4.5. *The types H and $\|A\|_n$ are equivalent.*

At the same time, we have the following:

Lemma 4.6 (Universal property of H)**.** *For any $(n+1)$-type B, the type of functions $H \to B$ is equivalent to*

$$\begin{aligned}&\Sigma\,(f : A \to B)\,.\\ &\Sigma\,(e : \Pi_{a,b:A}\|a = b\|_{n-1} \to f(a) = f(b))\,.\\ &\quad (d : \Pi_{a:A}\mathsf{refl}_{f(a)} = e(a, a, |\mathsf{refl}_a|)).\end{aligned} \tag{43}$$

Proof sketch. The proof of deriving this form of universal property from the induction principle is standard. The map from $H \to B$ into the stated type is more or less composition with the constructors; for any $k : H \to B$, we get

$$(f, e, d) :\equiv \big(k \circ \eta\,,\ \mathsf{ap}_k \circ \epsilon\,,\ \lambda a.\mathsf{ap}_{\mathsf{ap}_k}(\delta(a))\big)\,. \tag{44}$$

The map in the other direction is exactly the recursion principle of H. That they are mutually inverse corresponds to the computation (β) rule respectively the uniqueness (η) rule of H. □

Finally, we can complete the second proof of our main result:

"HIT proof" of Lemma 2.3. We do induction on n. The base case ($n \equiv -1$) is, as before, just what we have discussed in Section 2. For higher n, we have the following chain of equivalences:

$$\|A\|_n \to B \tag{45}$$

(by Corollary 4.5)

$$\simeq \quad H \to B \tag{46}$$

(by Lemma 4.6)

$$\begin{aligned}\simeq \quad &\Sigma\,(f : A \to B)\,.\,\Sigma\,(e : \Pi_{a,b:A}\|a = b\|_{n-1} \to f(a) = f(b))\,.\\ &\qquad \big(\Pi_{a:A}\mathsf{refl}_{f(a)} = e(a, a, |\mathsf{refl}_a|)\big)\end{aligned} \tag{47}$$

(by "inverse path induction")

$$\begin{aligned}\simeq \quad &\Sigma\,(f : A \to B)\,.\,\Sigma\,(e : \Pi_{a,b:A}\|a = b\|_{n-1} \to f(a) = f(b))\,.\\ &\qquad \big(\Pi_{a,b:A}\Pi_{p:a=b}\mathsf{ap}_f p = e(a, b, |p|)\big)\end{aligned} \tag{48}$$

(by the distributivity law)

$$\begin{aligned}\simeq \quad &\Sigma\,(f : A \to B)\,.\,\Pi_{a,b:A}\big(\Sigma\,(e' : \|a = b\|_{n-1} \to f(a) = f(b))\,.\\ &\qquad \Pi_{p:a=b}\mathsf{ap}_f p = e'(|p|)\big)\end{aligned} \tag{49}$$

Now we exchange e' by $(e_1, e_2) :\equiv \mathfrak{c}_{n-1}(e')$ using the induction hypothesis, and thus we need to apply $\mathfrak{c}_{n-1}^{-1}$ to that term in the last component. Fortunately, it follows from the definition of $\mathfrak{c}_{n-1}$ that $\_ \circ \mathfrak{c}_{n-1} \equiv \pi_1 \circ |-|$, hence we can replace $e'(|p|)$ with simply $e_1(p)$:

$$\begin{aligned}\simeq \quad &\Sigma\,(f : A \to B)\,.\,\Pi_{a,b:A}\big(\Sigma\,(e_1 : a = b \to f(a) = f(b))\,.\,\Sigma\,\big(e_2 : \Pi_{p:a=b}\mathsf{isNull}(\mathsf{ap}^n_{e_1,p})\big)\,.\\ &\qquad \big(\Pi_{p:a=b}\mathsf{ap}_f p = e_1(p)\big)\big)\end{aligned} \tag{50}$$

The term e_1 and the very last (unnamed) component form a singleton and can be removed:

$$\simeq \quad \Sigma\,(f : A \to B)\,.\,\left(\Pi_{a,b:A}\Pi_{p:a=b}\mathsf{isNull}(\mathsf{ap}^n_{\mathsf{ap}_f,p})\right) \tag{51}$$

(by "path induction")

$$\simeq \quad \Sigma\,(f : A \to B)\,.\,\left(\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^n_{\mathsf{ap}_f,\mathsf{refl}_{f(a)}})\right) \tag{52}$$

(as $\mathsf{ap}^n_{\mathsf{ap}_f,\mathsf{refl}_a}$ is the same as $\mathsf{ap}^{n+1}_{f,a}$ – the footnote on page 7 applies)

$$\simeq \quad \Sigma\,(f : A \to B)\,.\,\left(\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,\mathsf{refl}_a})\right). \tag{53}$$

Finally, we need to check that the constructed equivalence is indeed the canonical function $\mathfrak{c}_n$. Fortunately, the second (and more involved) part $\Pi_{a:A}\mathsf{isNull}(\mathsf{ap}^{n+1}_{f,\mathsf{refl}_a})$ is propositional. It is therefore enough to check that any map $g : \|A\|_n \to B$ gets, by the constructed equivalence, mapped to a pair in (53) of which *the first component* is $g \circ |-|$. But the first component is constructed in the very first step, where Lemma 4.6 is applied, and, looking at the proof of Lemma 4.6, it is indeed simply composition with $|-|$. □

5. A Sample Application: Set-Based Groupoids

A set-theoretic ω-groupoid has, in the "globular" formulation, ω-many levels: At level 0, it has a collection of objects (or 0-cells); for any two objects, it has a collection of 1-morphisms (1-cells); for any two 1-morphisms, there is a collection of 2-morphisms (2-cells), and so on. As recalled in the introduction, types indeed are such ω-groupoids meta-theoretically. It is intuitive to ask how much of this can be internalised. Defining a weak ω-groupoid in type theory is already very hard [2, 3]: one would want a 0-type (i.e. a *set*) A_0 of 0-cells, a *set* A_1 of 1-cells which is indexed twice over A_0, and so on. Even if one has such a definition at hand, it is implausible to expect that one can define the "fundamental ω-groupoid" of a type. As Altenkirch, Li and Rypacek [2] mention, they are unable to construct such an ω-groupoid, which in their terminology is called $\mathsf{Id}\omega$. The Ph.D. thesis of the second-named author of the current paper includes a precise negative statement [9, Sec. 9.4.1] which shows that a construction in the sense of [2] is impossible in all non-trivial cases. The argument given there indicates that a fundamental reason why we cannot even define A_1 is that we want A_1 to be indexed twice over A_0.

However, we know that the whole higher structure of types is in some sense determined by the loop spaces, as opposed to the path spaces. It seems therefore reasonable to consider a more modest variation where we index A_1 only once over A_0, with the intention that $A_1(a_0)$ represents the loop space over a_0. This has the further advantage that we can assume that A_0 is $\|A\|_0$; with double-indexed A_1, it would be possible that elements $a, b : A_0$ are not equal in A_0, but "made equal" by an element of $A_1(a, b)$. As a further simplification, we only consider the question whether a type can be represented in two levels, i.e. with $A_0 \equiv \|A\|_0$ and A_1.

Definition 5.1. We call a type A *set-based representable* if the function

$$\omega_A : A \to \mathcal{U} \tag{54}$$

$$\omega_A(a) :\equiv (a = a) \tag{55}$$

factors through $\|A\|_0$, i.e. if there is a single-indexed family $A_1 : \|A\|_0 \to \mathcal{U}$ of types which, for all $a : A$, satisfies $A_1(|a|) \simeq (a =_A a)$.

We also define the following simple notion:

Definition 5.2. We say that a type A has *loop spaces with braidings* if, for all $a : A$ and $p, q : a = a$, we have $p \cdot q = q \cdot p$.

Examples of types which have loop spaces with braidings are sets (for which the condition is trivial), and, more interestingly, loop spaces themselves.

Theorem 5.3. *Every* 1*-type whose loop spaces have braidings is set-based representable.*

Proof. As A is a 1-type, the function (54) takes sets as values; that is, in this case, we can assume that ω_A is of type $A \to \mathcal{U}^0$. Using that $\mathcal{U}^0$ is a 1-type [14, Thm. 7.1.11], we may apply Theorem 2.1 with $n \equiv 0$. We need to show that, for a fixed $a : A$, the function

$$\mathsf{ap}_{\omega_A,a} : \Omega_t(A,a) \to \Omega_t(\mathcal{U}, a = a) \tag{56}$$

is null. But $\mathsf{ap}_{\omega_A}(p)$ induces a *function* of type $(a = a) \to (a = a)$ (via the function that is called $\mathsf{idtoeqv}$ in [14], and projection), and by univalence, it is enough to show that this function does not depend on p. We claim that this function maps $q : a = a$ to $p^{-1} \cdot q \cdot p$. An easy way to prove this claim is considering the more general version of ap_{ω_A} that works on any path spaces (instead of loop spaces), and then doing path induction on p. Clearly, the braiding on $a = a$ is exactly what we need to justify that $p^{-1} \cdot q \cdot p$ does not depend on p. □

6. The Big Picture: Solved and Unsolved Cases

The "ordinary" universal property of the n-truncation can be recovered easily from Theorem 2.1. If, under the conditions of the statement, B is not only $(n+1)$-, but even n-truncated, the type $\Pi_{a:A}\mathsf{isNull}\left(\mathsf{ap}^{n+1}_{f,a}\right)$ becomes contractible, and the theorem says precisely that functions $A \to B$ are the same as functions $\|A\|_n \to B$, via composition with $|-|$. Theorem 2.1 is thus stronger than the "ordinary" universal property. However, we weaken the condition on B by only one single level, while [8] weakens it by arbitrary many levels, but only for the propositional truncation.

Of course, the general question is: What is the universal property of $\|A\|_n$ with respect to m-types, i.e. how can we construct a map $\|A\|_n \to B$ for some m-type B? Put differently, given a function $f : A \to B$, how can we (by only imposing conditions on f, not on A or B) ensure that f factors through $\|A\|_n$? Figure 2 illustrates the current progress on this question. As indicated, the question is trivial if m is not greater than n. Two other families of cases are solved, those with $m \equiv n+1$ by the current paper, and $n \equiv -1$ by [8]. Note that the latter is not internalised in the way that the result of the current paper is, and it is not to be expected that an internalisation is possible in the considered type theory; and further, the case $n \equiv -1$, $m \equiv \infty$ (meaning that there is no condition at all on B) is solved, but only under the assumption of Reedy ω^{op}-limits.

The (probably) simplest case that is left open is the case $n \equiv 0$, $m \equiv 2$. So, let us consider a function $f : A \to B$, where B is 2-truncated. Which conditions do we have to impose on f to conclude that it factors through $\|A\|_0$? As is easy to show, if f factors through the 0-truncation, then ap_f factors through the (-1)-truncation. The necessary conditions for the latter have been worked out in [8], and we could thus try to impose them on ap_f (at all points). However, this does not work. In one aspect, the propositional truncation is a special case that is actually *harder* than the higher truncations, intuitively because loop spaces are always pointed[10] which we have already made use of in the definition of isNull. It turns out that in this "pointed" case one can get all these coherences (which make the result of [8] hard) for free. Instead, the higher groupoid structure of loop spaces induces a different sort of coherence problem. For example, it certainly *is* necessary that, for any $a : A$

[10]This seems to correspond to the fact that the zeroth homotopy "group" is not a group, and does therefore not have a canonical element, which seems to occasionally make this special case harder in traditional topology as well.

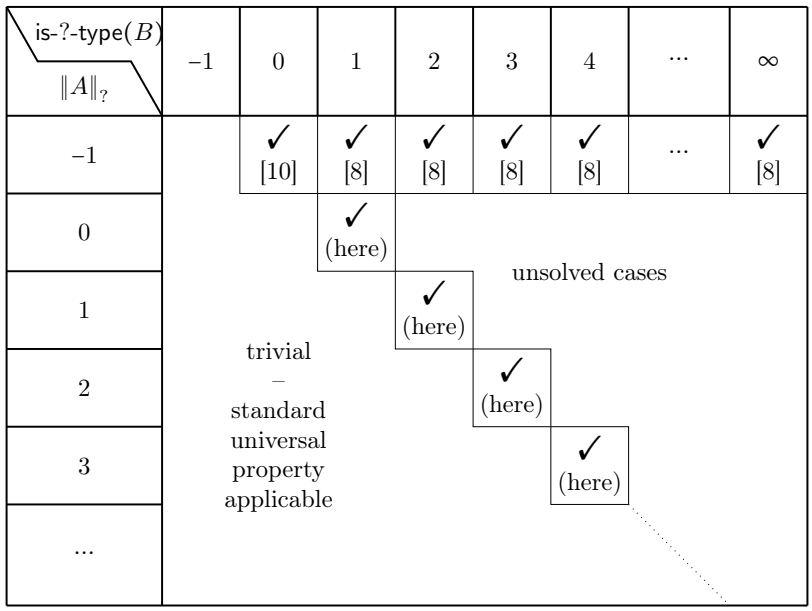

FIGURE 2. The universal property of $\|A\|_?$ with respect to ?-types: trivial, solved, and open cases

and $p : a = a$, there is a proof $c_{a,p} : \mathsf{ap}_{f,a}(p) = \mathsf{refl}_{f(a)}$. From $c_{a,p}$, we can construct a proof that $\mathsf{ap}_{a,f}(p \cdot p)$ equals $\mathsf{refl}_{f(a)}$, using functoriality of $\mathsf{ap}_{f,a}$. If we want the family c to be "fully coherent", we have to force this proof to be the same as $c_{a,p \cdot p}$. The work [8] concludes with a precise conjecture of how *all* the required coherence conditions can be captured in the general case. At this time, it is unknown whether this can be used to fill in the missing parts of Figure 2.

Acknowledgements. We would like to thank Thorsten Altenkirch and Christian Sattler for fruitful discussions. The second-named author is grateful for the opportunity to discuss some of the ideas that have led to this paper with participants of several events, including the *Institut Henri Poincaré thematic trimester*. We further want to acknowledge that Michael Shulman has made the connection with the Rezk completion precise, and we thank the anonymous reviewers for their reports that have helped us improving this paper.

REFERENCES

[1] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science (MSCS)*, pages 1–30, Jan 2015.

[2] Thorsten Altenkirch, Nuo Li, and Ondrej Rypacek. Some constructions on ω-groupoids. In *Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, 2014.

[3] Thorsten Altenkirch and Ondrej Rypacek. A syntactical approach to weak ω-groupoids. In *Computer Science Logic (CSL)*, pages 16–30, 2012.

[4] Steve Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

[5] Paolo Capriotti. Higher lenses. Blog post at `homotopytypetheory.org`, 29 Apr 2014.

[6] Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of higher truncations (Agda formalisation), Apr 2015. Available at `https://github.com/pcapriotti/agda-base/tree/trunc/hott/truncation`.

[7] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith.

Implementing Mathematics with the NuPRL Proof Development System. Prentice-Hall, NJ, 1986.

[8] Nicolai Kraus. The general universal property of the propositional truncation. *ArXiv e-prints*, Nov 2014. To appear in the post-proceedings of TYPES'14.

[9] Nicolai Kraus. *Truncation Levels in Homotopy Type Theory*. PhD thesis, School of Computer Science, University of Nottingham, Nottingham, UK, 2015.

[10] Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf type theory. Submitted, 2014.

[11] Nicolai Kraus and Christian Sattler. Higher homotopies in a hierarchy of univalent universes. *ACM Transactions on Computational Logic (TOCL)*, 16(2):18:1–18:12, April 2015.

[12] Peter LeFanu Lumsdaine. Weak omega-categories from intensional type theory. In *Typed Lambda Calculi and Applications (TLCA)*, pages 172–187. Springer-Verlag, 2009.

[13] Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, pages 1–75, Jan 2015.

[14] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `homotopytypetheory.org/book`, Institute for Advanced Study, first edition, 2013.

[15] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

Part II

Type-based Totality Checking

Chapter 3

A Formalized Proof of Strong Normalization for Guarded Recursive Types

A Formalized Proof of Strong Normalization for Guarded Recursive Types (Long Version)

Andreas Abel and Andrea Vezzosi

Computer Science and Engineering, Chalmers and Gothenburg University,
Rännvägen 6, 41296 Göteborg, Sweden
andreas.abel@gu.se, vezzosi@chalmers.se

Abstract. We consider a simplified version of Nakano's guarded fixed-point types in a representation by infinite type expressions, defined coinductively. Small-step reduction is parametrized by a natural number "depth" that expresses under how many guards we may step during evaluation. We prove that reduction is strongly normalizing for any depth. The proof involves a typed inductive notion of strong normalization and a Kripke model of types in two dimensions: depth and typing context. Our results have been formalized in Agda and serve as a case study of reasoning about a language with coinductive type expressions.

1 Introduction

In untyped lambda calculus, fixed-point combinators can be defined using self-application. Such combinators can be assigned recursive types, albeit only negative ones. Since such types introduce logical inconsistency, they are ruled out in Martin-Löf Type Theory and other systems based on the Curry-Howard isomorphism. Nakano (2000) introduced *a modality for recursion* that allows a stratification of negative recursive types to recover consistency. In essence, each negative recursive occurrence needs to be *guarded* by the modality; this coined the term *guarded recursive types* (Birkedal and Møgelberg, 2013).[1] Nakano's modality has found applications in functional reactive programming (Krishnaswami and Benton, 2011b) where it is referred to as *later* modality.

While Nakano showed that every typed term has a weak head normal form, in this paper we prove *strong normalization* for our variant $\lambda^{\blacktriangleright}$ of Nakano's calculus. To this end, we make the introduction rule for the later modality explicit in the terms by a constructor next, following Birkedal and Møgelberg (2013) and Atkey and McBride (2013). By allowing reduction under finitely many nexts, we establish termination irrespective of the reduction strategy. Showing strong normalization of $\lambda^{\blacktriangleright}$ is a first step towards an operationally well-behaved type theory with guarded recursive types, for which Birkedal and Møgelberg (2013) have given a categorical model.

Our proof is fully formalized in the proof assistant Agda (2014) which is based on intensional Martin-Löf Type Theory. [2] One key idea of the formalization is to represent

[1] Not to be confused with *Guarded Recursive Datatype Constructors* (Xi et al., 2003).

[2] A similar proof could be formalized in other systems supporting mixed induction-coinduction, for instance, in Coq.

the recursive types of $\lambda^{\blacktriangleright}$ as infinite type expressions in form of a coinductive definition. For this, we utilize Agda's new *copattern* feature (Abel et al., 2013). The set of strongly normalizing terms is defined inductively by distinguishing on the shape of terms, following van Raamsdonk et al. (1999) and Joachimski and Matthes (2003). The first author has formalized this technique before in Twelf (Abel, 2008); in this work we extend these results by a proof of equivalence to the standard notion of strong normalization.

Due to space constraints, we can only give a sketch of the formalization; a longer version and the full Agda proofs are available online (Abel and Vezzosi, 2014). This paper is extracted from a literate Agda file; all the colored code in displays is necessarily type-correct.

2 Guarded Recursive Types and Their Semantics

Nakano's type system (2000) is equipped with subtyping, but we stick to a simpler variant without, a simply-typed version of Birkedal and Møgelberg (2013), which we shall call $\lambda^{\blacktriangleright}$. Our rather minimal grammar of types includes product $A \times B$ and function types $A \to B$, delayed computations $\blacktriangleright A$, variables X and explicit fixed-points μXA.

$$A,B,C ::= A \times B \mid A \to B \mid \blacktriangleright A \mid X \mid \mu XA$$

Base types and disjoint sum types could be added, but would only give breadth rather than depth to our formalization. As usual, a dot after a bound variable shall denote an opening parenthesis that closes as far to the right as syntactically possible. Thus, $\mu X.X \to X$ denotes $\mu X\,(X \to X)$, while $\mu XX \to X$ denotes $(\mu X.X) \to X$ (with a free variable X).

Formation of fixed-points μXA is subject to the side condition that X is guarded in A, i. e., X appears in A only under a *later* modality $\blacktriangleright$. This rules out all unguarded recursive types like $\mu X.A \times X$ or $\mu X.X \to A$, but allows their variants $\mu X.\blacktriangleright(A \times X)$ and $\mu X.A \times \blacktriangleright X$, and $\mu X.\blacktriangleright(X \to A)$ and $\mu X.\blacktriangleright X \to A$. Further, fixed-points give rise to an equality relation on types induced by $\mu XA = A[\mu XA/X]$.

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

$$\frac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\, t : A_1} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \mathsf{snd}\, t : A_2}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \blacktriangleright A} \qquad \frac{\Gamma \vdash t : \blacktriangleright(A \to B) \qquad \Gamma \vdash u : \blacktriangleright A}{\Gamma \vdash t * u : \blacktriangleright B} \qquad \frac{\Gamma \vdash t : A \qquad A = B}{\Gamma \vdash t : B}$$

Fig. 1. Typing rules.

Terms are lambda-terms with pairing and projection plus operations that witness *applicative functoriality* of the later modality (Atkey and McBride, 2013).

$$t,u ::= x \mid \lambda x t \mid t\,u \mid (t_1,t_2) \mid \mathsf{fst}\,t \mid \mathsf{snd}\,t \mid \mathsf{next}\,t \mid t * u$$

Figure 1 recapitulates the static semantics. The dynamic semantics is induced by the following *contractions*:

$$\begin{array}{ll} (\lambda x.\,t)\,u & \mapsto\ t[u/x] \\ \mathsf{fst}\,(t_1,t_2) & \mapsto\ t_1 \\ \mathsf{snd}\,(t_1,t_2) & \mapsto\ t_2 \\ (\mathsf{next}\,t) * (\mathsf{next}\,u) & \mapsto\ \mathsf{next}\,(t\,u) \end{array}$$

If we conceive our small-step reduction relation $\longrightarrow$ as the compatible closure of $\mapsto$, we obtain a non-normalizing calculus, since terms like $\Omega = \omega\,(\mathsf{next}\,\omega)$ with $\omega = (\lambda x.\,x * (\mathsf{next}\,x))$ are typeable.[3] Unrestricted reduction of Ω is non-terminating: $\Omega \longrightarrow \mathsf{next}\,\Omega \longrightarrow \mathsf{next}\,(\mathsf{next}\,\Omega) \longrightarrow \ldots$ If we let next act as delay operator that blocks reduction inside, we regain termination. In general, we preserve termination if we only look under delay operators up to a certain depth. This can be made precise by a family $\longrightarrow_n$ of reduction relations indexed by a depth $n \in \mathbb{N}$, see Figure 2.

$$\frac{t \mapsto t'}{t \longrightarrow_n t'} \qquad \frac{t \longrightarrow_n t'}{\lambda x.\,t \longrightarrow_n \lambda x.\,t'} \qquad \frac{t \longrightarrow_n t'}{t\,u \longrightarrow_n t'\,u} \qquad \frac{u \longrightarrow_n u'}{t\,u \longrightarrow_n t\,u'}$$

$$\frac{t \longrightarrow_n t'}{(t,u) \longrightarrow_n (t',u)} \qquad \frac{u \longrightarrow_n u'}{(t,u) \longrightarrow_n (t,u')} \qquad \frac{t \longrightarrow_n t'}{\mathsf{fst}\,t \longrightarrow_n \mathsf{fst}\,t'} \qquad \frac{t \longrightarrow_n t'}{\mathsf{snd}\,t \longrightarrow_n \mathsf{snd}\,t'}$$

$$\boxed{\frac{t \longrightarrow_n t'}{\mathsf{next}\,t \longrightarrow_{n+1} \mathsf{next}\,t'}} \qquad \frac{t \longrightarrow_n t'}{t * u \longrightarrow_n t' * u} \qquad \frac{u \longrightarrow_n u'}{t * u \longrightarrow_n t * u'}$$

Fig. 2. Reduction

We should note that for a fixed depth n the relation $\longrightarrow_n$ is not confluent. In fact the term $(\lambda z.\,\mathsf{next}^{n+1}\,z)(\mathsf{fst}\,(u,t))$ reduces to two different normal forms, $\mathsf{next}^{n+1}\,(\mathsf{fst}\,(u,t))$ and $\mathsf{next}^{n+1}\,u$. We could remedy this situation by making sure we never hide redexes under too many applications of next and instead store them in an explicit substitution where they would still be accessible to $\longrightarrow_n$. Our problematic terms would then look like $\mathsf{next}^n\,((\mathsf{next}\,z)[\mathsf{fst}\,(u,t)/z])$ and $\mathsf{next}^n\,((\mathsf{next}\,z)[u/z])$ and the former would reduce to the latter. However, we are not bothered by the non-confluence since our semantics at level n (see below) does not distinguish between $\mathsf{next}^{n+1}u$ and $\mathsf{next}^{n+1}u'$ (as in $u' = \mathsf{fst}\,(u,t)$); neither u nor u' is required to terminate if buried under more than n nexts.

To show termination, we interpret types as sets $\mathscr{A},\mathscr{B},\mathscr{C}$ of depth-n strongly normalizing terms. We define semantic versions $[\![\times]\!]$, $[\![\to]\!]$, and $[\![\blacktriangleright]\!]$ of product, function

[3] $\vdash \Omega : A$ with $A = \mu X(\blacktriangleright X)$. To type ω, we use $x : \mu Y(\blacktriangleright(Y \to A))$.

space, and delay type constructor, plus a terminal (=largest) semantic type $[\![\top]\!]$. Then the interpretation $[\![A]\!]_n$ of closed type A at depth n can be given recursively as follows, using the Kripke construction at function types:

$$
\begin{array}{lll}
[\![A \times B]\!]_n & = [\![A]\!]_n \; [\![\times]\!] \; [\![B]\!]_n & \mathscr{A} \; [\![\times]\!] \; \mathscr{B} = \{t \mid \mathsf{fst}\, t \in \mathscr{A} \text{ and } \mathsf{snd}\, t \in \mathscr{B}\} \\
[\![A \to B]\!]_n & = \bigcap_{n' \le n} ([\![A]\!]_{n'} \; [\![\to]\!] \; [\![B]\!]_{n'}) & \mathscr{A} \; [\![\to]\!] \; \mathscr{B} = \{t \mid t\, u \in \mathscr{B} \text{ for all } u \in \mathscr{A}\} \\
[\![\blacktriangleright A]\!]_0 & = [\![\blacktriangleright]\!] \; [\![\top]\!] & [\![\top]\!] = \{t \mid t \text{ term}\} \\
[\![\blacktriangleright A]\!]_{n+1} & = [\![\blacktriangleright]\!] [\![A]\!]_n & [\![\blacktriangleright]\!] \, \mathscr{A} = \overline{\{\mathsf{next}\, t \mid t \in \mathscr{A}\}} \\
[\![\mu X A]\!]_n & = [\![A[\mu X A / X]]\!]_n & (\overline{\mathscr{A}} \text{ is weak head expansion closure of } \mathscr{A})
\end{array}
$$

Due to the last equation (μ), the type interpretation is ill-defined for unguarded recursive types. However, for guarded types we only return to the fixed-point case after we have passed the case for $\blacktriangleright$, which decreases the index n. More precisely, $[\![A]\!]_n$ is defined by lexicographic induction on $(n, \mathsf{size}(A))$, where $\mathsf{size}(A)$ is the number of type constructor symbols $(\times, \to, \mu)$ that occur *unguarded* in A.

While all this sounds straightforward at an informal level, formalization of the described type language is quite hairy. For one, we have to enforce the restriction to well-formed (guarded) types. Secondly, our type system contains a conversion rule, getting us into the vincinity of dependent types which are still a challenge to a completely formal treatment (McBride, 2010). Our first formalization attempt used kinding rules for types to keep track of guardedness for formation of fixed-point, and a type equality relation, and building on this, inductively defined well-typed terms. However, the complexity was discouraging and lead us to a much more economic representation of types, which is described in the next section.

3 Formalized Syntax

In this section, we discuss the formalization of types, terms, and typing of $\lambda^{\blacktriangleright}$ in Agda. It will be necessary to talk about meta-level types, i. e., Agda's types, thus, we will refer to $\lambda^{\blacktriangleright}$'s type constructors as $\hat{\times}$, $\hat{\to}$, $\hat{\blacktriangleright}$, and $\hat{\mu}$.

3.1 Types Represented Coinductively

Instead of representing fixed-points as syntactic construction on types, which would require a non-trivial equality on types induced by $\hat{\mu} XA = A[\hat{\mu} XA/X]$, we use *meta-level* fixed-points, i. e., Agda's recursion mechanism.[4] Extensionally, we are implementing *infinite type expressions* over the constructors $\hat{\times}$, $\hat{\to}$, and $\hat{\blacktriangleright}$. The guard condition on recursive types then becomes an instance of Agda's "guard condition", i. e., the condition the termination checker imposes on recursive programs.

[4] An alternative to get around the type equality problem would be iso-recursive types, i. e., with term constructors for folding and unfolding of $\hat{\mu} XA$. However, we would still have to implement type variables, binding of type variables, type substitution, lemmas about type substitution etc.

Viewed as infinite expressions, guarded types are regular trees with an infinite number of $\hat{\blacktriangleright}$-nodes on each infinite path. This can be expressed as the mixed coinductive(ν)-inductive(μ) (meta-level) type

$$\nu X \mu Y.\ (Y \times Y) + (Y \times Y) + X.$$

The first summand stands for the binary constructor $\hat{\times}$, the second for $\hat{\to}$, and the third for the unary $\hat{\blacktriangleright}$. The nesting of a least-fixed point (μ) inside a greatest fixed-point (ν) ensures that on each path, we can only take alternatives $\hat{\times}$ and $\hat{\to}$ a finite number of times before we have to choose the third alternative $\hat{\blacktriangleright}$ and restart the process.

In Agda 2.4, we represent this mixed coinductive-inductive type by a datatype Ty (inductive component) mutually defined with a record ∞Ty (coinductive component).

```
mutual
  data Ty : Set where
    _×̂_  : (a b : Ty)  → Ty
    _→̂_  : (a b : Ty)  → Ty
    ▶̂_   : (a∞ : ∞Ty)  → Ty

  record ∞Ty : Set where
    coinductive
    constructor delay
    field       force : Ty
```

While the arguments a and b of the infix constructors $\hat{\times}$ and $\hat{\to}$ are again in Ty, the prefix constructor $\hat{\blacktriangleright}$ expects and argument $a\infty$ in ∞Ty, which is basically a wrapping[5] of Ty. The functions delay and force convert back and forth between Ty and ∞Ty so that both types are valid representations of the set of types of $\lambda^{\blacktriangleright}$.

$$\begin{aligned} \mathsf{delay} &: \mathsf{Ty} \to \infty\mathsf{Ty} \\ \mathsf{force} &: \infty\mathsf{Ty} \to \mathsf{Ty} \end{aligned}$$

However, since ∞Ty is declared coinductive, its inhabitants are not evaluated until forced. This allows us to represent infinite type expressions, like $\mathsf{top} = \hat{\mu}X(\hat{\blacktriangleright}X)$.

```
top : ∞Ty
force top = ▶̂ top
```

Technically, top is defined by *copattern* matching (Abel et al., 2013); top is uniquely defined by the value of its only field, force top, which is given as $\hat{\blacktriangleright}$ top. Agda will use the given equation for its internal normalization procedure during type-checking. Alternatively, we could have tried to define top : Ty by top = $\hat{\blacktriangleright}$ delay top. However, Agda will rightfully complain here since rewriting with this equation would keep expanding top forever, thus, be non-terminating. In contrast, rewriting with the original equation is terminating since at each step, one application of force is removed.

The following two defined type constructors will prove useful in the definition of well-typed terms to follow.

```
▶_ : Ty → Ty
▶ a = ▶̂ delay a
```

[5] Similar to a `newtype` in the functional programming language Haskell.

```
_⇒_ : (a∞ b∞ : ∞Ty) → ∞Ty
force (a∞ ⇒ b∞) = force a∞ -→ force b∞
```

3.2 Well-typed terms

Instead of a raw syntax and a typing relation, we represent well-typed terms directly by an inductive family (Dybjer, 1994). Our main motivation for this choice is the beautiful inductive definition of strongly normalizing terms to follow in Section 5. Since it relies on a classification of terms into the three shapes *introduction*, *elimination*, and *weak head redex*, it does not capture all strongly normalizing raw terms, in particular "junk" terms such as fst (λxx). Of course, statically well-typed terms come also at a cost: for almost all our predicates on terms we need to show that they are natural in the typing context, i. e., closed under well-typed renamings. This expense might be compensated by the extra assistance Agda can give us in proof construction, which is due to the strong constraints on possible solutions imposed by the rich typing.

Our encoding of well-typed terms follows closely Altenkirch and Reus (1999); McBride (2006); Benton et al. (2012). We represent typed variables $x : \mathsf{Var}\ \Gamma\ a$ by de Brujin indices, i. e., positions in a typing context $\Gamma : \mathsf{Cxt}$, which is just a list of types.

```
Cxt = List Ty

data Var : (Γ : Cxt) (a : Ty) → Set where
  zero : ∀{Γ a}     → Var (a :: Γ) a
  suc  : ∀{Γ a b} (x : Var Γ a) → Var (b :: Γ) a
```

Arguments enclosed in braces, such as Γ, a, and b in the types of the constructors zero and suc, are hidden and can in most cases be inferred by Agda. If needed, they can be passed in braces, either as positional arguments (e. g., $\{\Delta\}$) or as named arguments (e. g., $\{\Gamma = \Delta\}$). If $\forall$ prefixes bindings in a function type, the types of the bound variables may be omitted. Thus, $\forall\{\Gamma\ a\} \to \mathrm{A}$ is short for $\{\Gamma : \mathsf{Cxt}\}\{a : \mathsf{Ty}\} \to \mathrm{A}$.

Terms $t : \mathsf{Tm}\ \Gamma\ a$ are indexed by a typing context Γ and their type a, guaranteeing well-typedness and well-scopedness. In the following data type definition, $\mathsf{Tm}\ (\Gamma : \mathsf{Cxt})$ shall mean that all constructors uniformly take Γ as their first (hidden) argument.

```
data Tm (Γ : Cxt) : (a : Ty) → Set where
  var  : ∀{a}     (x : Var Γ a)                                  → Tm Γ a
  abs  : ∀{a b}   (t : Tm (a :: Γ) b)                            → Tm Γ (a -→ b)
  app  : ∀{a b}   (t : Tm Γ (a -→ b)) (u : Tm Γ a)               → Tm Γ b
  pair : ∀{a b}   (t : Tm Γ a)        (u : Tm Γ b)               → Tm Γ (a ^× b)
  fst  : ∀{a b}   (t : Tm Γ (a ^× b))                            → Tm Γ a
  snd  : ∀{a b}   (t : Tm Γ (a ^× b))                            → Tm Γ b
  next : ∀{a∞}    (t : Tm Γ (force a∞))                          → Tm Γ (▶̂ a∞)
  _*_  : ∀{a∞ b∞} (t : Tm Γ (▶̂(a∞ ⇒ b∞))) (u : Tm Γ (▶̂ a∞))    → Tm Γ (▶̂ b∞)
```

The most natural typing for next and $*$ would be using the defined $\blacktriangleright\_ : \mathsf{Ty} \to \mathsf{Ty}$:

```
  next : ∀{a}   (t : Tm Γ a)                          → Tm Γ (▶ a)
  _*_  : ∀{a b} (t : Tm Γ (▶(a -→ b))) (u : Tm Γ (▶ a)) → Tm Γ (▶ b)
```

However, this would lead to indices like ▶̂ delay a and unification problems Agda cannot solve, since matching on a coinductive constructor like delay is forbidden—it can

lead to a loss of subject reduction (McBride, 2009). The chosen alternative typing, which parametrizes over $a\infty\ b\infty : \infty\mathsf{Ty}$ rather than $a\ b : \mathsf{Ty}$, works better in practice.

3.3 Type Equality

Although our coinductive representation of $\lambda^{\blacktriangleright}$ types saves us from type variables, type substitution, and fixed-point unrolling, the question of type equality is not completely settled. The propositional equality $\equiv$ of Martin-Löf Type Theory is intensional in the sense that only objects with the same *code* (modulo definitional equality) are considered equal. Thus, $\equiv$ is adequate only for finite objects (such as natural numbers and lists) but not for infinite objects like functions, streams, or $\lambda^{\blacktriangleright}$ types.

However, we can define extensional equality or *bisimulation* on Ty as a mixed coinductive-inductive relation $\cong/\infty\cong$ that follows the structure of $\mathsf{Ty}/\infty\mathsf{Ty}$ (hence, we reuse the constructor names $\hat{\times}$, $\hat{\to}$, and $\hat{\blacktriangleright}$).

```
mutual
  data _≅_ : (a b : Ty) → Set where
    _×̂_ : ∀{a a' b b'} (a≅ : a ≅ a') (b≅ : b ≅ b') → (a ×̂ b) ≅ (a' ×̂ b')
    _→̂_ : ∀{a a' b b'} (a≅ : a' ≅ a) (b≅ : b ≅ b') → (a →̂ b) ≅ (a' →̂ b')
    ▶̂_  : ∀{a∞ b∞}     (a≅ : a∞ ∞≅ b∞)            → ▶̂ a∞ ≅ ▶̂ b∞

  record _∞≅_ (a∞ b∞ : ∞Ty) : Set where
    coinductive
    constructor ≅delay
    field       ≅force : force a∞ ≅ force b∞
```

Ty-equality is indeed an equivalence relation (we omit the standard proof).

```
≅refl  : ∀{a}     → a ≅ a
≅sym   : ∀{a b}   → a ≅ b → b ≅ a
≅trans : ∀{a b c} → a ≅ b → b ≅ c → a ≅ c
```

However, unlike for $\equiv$ we do not get a generic substitution principle for $\cong$, but have to prove it for any function and predicate on Ty. In particular, we have to show that we can cast a term in $\mathsf{Tm}\ \Gamma\ a$ to $\mathsf{Tm}\ \Gamma\ b$ if $a \cong b$, which would require us to build type equality at least into $\mathsf{Var}\ \Gamma\ a$. In essence, this would amount to work with setoids across all our development, which would add complexity without strengthening our result. Hence, we fall for the shortcut:

It is consistent to postulate that bisimulation implies equality, similarly to the functional extensionality principle for function types. This lets us define the function cast to convert terms between bisimilar types.

```
postulate ≅-to-≡ : ∀ {a b} → a ≅ b → a ≡ b

cast : ∀{Γ a b} (eq : a ≅ b) (t : Tm Γ a) → Tm Γ b
```

We shall require cast in uses of functorial application, to convert a type $c\infty : \infty\mathsf{Ty}$ into something that can be forced into a function type.

```
▶app : ∀{Γ c∞ b∞ a} (eq : c∞ ∞≅ (delay a ⇒ b∞))
                    (t : Tm Γ (▶̂ c∞)) (u : Tm Γ (▶ a)) → Tm Γ (▶̂ b∞)
▶app eq t u = cast (▶̂ eq) t ∗ u
```

3.4 Examples

Following Nakano (2000), we can adapt the Y combinator from the untyped lambda calculus to define a guarded fixed point combinator:

$$\mathsf{fix} = \lambda f.\ (\lambda x.\ f\ (x \circledast \mathsf{next}\ x))\ (\mathsf{next}\ (\lambda x.\ f\ (x \circledast \mathsf{next}\ x))).$$

We construct an auxiliary type Fix a that allows safe self application, since the argument will only be available "later". This fits with the type we want for the fix combinator, which makes the recursive instance y in fix $(\lambda y.t)$ available only at the next time slot.

```
fix : ∀{Γ a} → Tm Γ ((▶ a ⇀ a) ⇀ a)

Fix_ : Ty → ∞Ty
force (Fix a) = ▶̂ Fix a ⇀ a

selfApp : ∀{Γ a} → Tm Γ (▶̂ Fix a) → Tm Γ (▶ a)
selfApp x = ▶app (≡delay ≡refl) x (next x)

fix = abs (app L (next L))
  where
    f = var (suc zero)
    x = var zero
    L = abs (app f (selfApp x))
```

Another standard example is the type of streams, which we can also define through corecursion.

```
mutual
  Stream : Ty → Ty
  Stream a = a ×̂ ▶̂ Stream∞ a

  Stream∞ : Ty → ∞Ty
  force (Stream∞ a) = Stream a

cons : ∀{Γ a} → Tm Γ a → Tm Γ (▶ Stream a) → Tm Γ (Stream a)
cons a s = pair a (cast (▶̂ (≡delay ≡refl)) s)

head : ∀{Γ a} → Tm Γ (Stream a) → Tm Γ a
head s = fst s

tail : ∀{Γ a} → Tm Γ (Stream a) → Tm Γ (▶ Stream a)
tail s = cast (▶̂ (≡delay ≡refl)) (snd s)
```

Note that tail returns a stream inside the later modality. This ensures that functions that transform streams have to be causal, i. e., can only have access to the first n elements of the input when producing the nth element of the output. A simple example is mapping a function over a stream.

```
mapS : ∀{Γ a b} → Tm Γ ((a ⇀ b) ⇀ (Stream a ⇀ Stream b))
```

Which is also better read with named variables.

$$\mathsf{mapS} = \lambda f.\ \mathsf{fix}\ (\lambda mapS.\ \lambda s.\ (f\ s, mapS \circledast \mathsf{tail}\ s))$$

4 Reduction

In this section, we describe the implementation of parametrized reduction $\longrightarrow_n$ in Agda. As a prerequisite, we need to define substitution, which in turn depends on renaming (Benton et al., 2012).

A *renaming* from context Γ to context Δ, written $\Delta \leq \Gamma$, is a mapping from variables of Γ to those of Δ of the same type a. The function rename lifts such a mapping to terms.

```
_≤_ : (Δ Γ : Cxt) → Set
_≤_ Δ Γ = ∀ {a} → Var Γ a → Var Δ a

rename : ∀ {Γ Δ : Cxt} {a : Ty} (η : Δ ≤ Γ) (x : Tm Γ a) → Tm Δ a
```

Building on renaming, we define well-typed parallel substitution. From this, we get the special case of substituting de Bruijn index 0.

```
subst0 : ∀ {Γ a b} → Tm Γ a → Tm (a :: Γ) b → Tm Γ b
```

Reduction $t \longrightarrow_n t'$ is formalized as the inductive family $t \langle n \rangle \Rightarrow \beta\ t'$ with four constructors β... representing the contraction rules and one congruence rule cong to reduce in subterms.

```
data _⟨_⟩⇒β_ {Γ} : ∀ {a} → Tm Γ a → ℕ → Tm Γ a → Set where

  β    : ∀ {n a b}{t : Tm (a :: Γ) b}{u}
       → app (abs t) u ⟨ n ⟩⇒β subst0 u t

  βfst : ∀ {n a b}{t : Tm Γ a}{u : Tm Γ b}
       → fst (pair t u) ⟨ n ⟩⇒β t

  βsnd : ∀ {n a b}{t : Tm Γ a}{u : Tm Γ b}
       → snd (pair t u) ⟨ n ⟩⇒β u

  β▶   : ∀ {n a∞ b∞}{t : Tm Γ (force a∞ ↛ force b∞)}{u : Tm Γ (force a∞)}
       → (next t ∗ next {a∞ = a∞} u) ⟨ n ⟩⇒β (next {a∞ = b∞} (app t u))

  cong : ∀ {n n' Δ a b t t' Ct Ct'}{C : NβCxt Δ Γ a b n n'}
       → (Ct  : Ct ≡ C [ t ])
       → (Ct' : Ct' ≡ C [ t' ])
       → (t⇒β : t ⟨ n ⟩⇒β t')
       → Ct ⟨ n' ⟩⇒β Ct'
```

The congruence rule makes use of shallow one hole contexts C, which are given by the following grammar

$$C ::= \lambda x\_ \mid \_u \mid t\_ \mid (t,\_) \mid (\_,u) \mid \mathsf{fst}\ \_ \mid \mathsf{snd}\ \_ \mid \mathsf{next}\ \_ \mid \_ * u \mid t * \_.$$

cong says that we can reduce a term, suggestively called Ct, to a term Ct', if (1) Ct decomposes into $C[t]$, a context C filled by t, and (2) Ct' into $C[t']$, and (3) t reduces to t'. As witnessed by relation $Ct \equiv C[t]$, context C : NβCxt $\Gamma\ \Delta\ a\ b\ n\ n'$ produces a term Ct : Tm $\Gamma\ b$ of depth n' if filled with a term t : Tm $\Delta\ a$ of depth n. The depth is unchanged except for the case next, which increases the depth by 1. Thus, $t \langle n \rangle \Rightarrow \beta\ t'$ can contract every subterm that is under at most n many nexts.

```
data NβCxt : (Δ Γ : Cxt) (a b : Ty) (n n' : ℕ) → Set where
  abs  : ∀{Γ n a b}                  → NβCxt (a :: Γ) Γ b (a ↛ b) n n
  appl : ∀{Γ n a b} (u : Tm Γ a)     → NβCxt Γ Γ (a ↛ b) b n n
```

```
appr  : ∀{Γ n a b} (t  : Tm Γ (a ⇀̂ b))      → NβCxt Γ Γ a b n n
pairl : ∀{Γ n a b} (u  : Tm Γ b)            → NβCxt Γ Γ a (a ×̂ b) n n
pairr : ∀{Γ n a b} (t  : Tm Γ a)            → NβCxt Γ Γ b (a ×̂ b) n n
fst   : ∀{Γ n a b}                          → NβCxt Γ Γ (a ×̂ b) a n n
snd   : ∀{Γ n a b}                          → NβCxt Γ Γ (a ×̂ b) b n n
next  : ∀{Γ n a∞}                           → NβCxt Γ Γ (force a∞) (▶̂ a∞) n (1 + n)
*l_   : ∀{Γ n a∞ b∞} (u : Tm Γ (▶̂ a∞))      → NβCxt Γ Γ (▶̂ (a∞ ⇒ b∞)) (▶̂ b∞) n n
*r_   : ∀{Γ n a∞ b∞}
        (t : Tm Γ (▶̂ (a∞ ⇒ b∞)))            → NβCxt Γ Γ (▶̂ a∞) (▶̂ b∞) n n


data _≡_[_] {n : ℕ} {Γ : Cxt} : {n' : ℕ} {Δ : Cxt} {b a : Ty} →
            Tm Γ b → NβCxt Δ Γ a b n n' → Tm Δ a → Set
```

5 Strong Normalization

Classically, a term is *strongly normalizing* (sn) if there's no infinite reduction sequence starting from it. Constructively, the tree of all the possible reductions from an sn term must be well-founded, or, equivalently, an sn term must be in the accessible part of the reduction relation. In our case, reduction $t\ \langle n\rangle{\Rightarrow}\beta\ t'$ is parametrized by a depth n, thus, we get the following family of sn-predicates.

```
data sn (n : ℕ) {a Γ} (t : Tm Γ a) : Set where
  acc : (∀ {t'} → t ⟨ n ⟩⇒β t' → sn n t') → sn n t
```

Van Raamsdonk et al. (1999) pioneered a more explicit characterization of strongly normalizing terms SN, namely the least set closed under introductions, formation of neutral (=stuck) terms, and weak head expansion. We adapt their technique from lambda-calculus to $\lambda^{\blacktriangleright}$; herein, it is crucial to work with well-typed terms to avoid junk like $\mathsf{fst}\,(\lambda x.x)$ which does not exist in pure lambda-calculus. To formulate a deterministic weak head evaluation, we make use of the *evaluation contexts* E : ECxt

$$E ::= \_\ u \mid \mathsf{fst}\ \_ \mid \mathsf{snd}\ \_ \mid \_ * u \mid (\mathsf{next}\ t) * \_.$$

Since weak head reduction does not go into introductions which include λ-abstraction, it does not go under binders, leaving typing context Γ fixed.

```
data ECxt (Γ : Cxt) : (a b : Ty) → Set
data _≅_[_] {Γ : Cxt} : {a b : Ty} → Tm Γ b → ECxt Γ a b → Tm Γ a → Set
```

$Et{\cong}E[t]$ witnesses the splitting of a term Et into evaluation context E and hole content t. A generalization of _≅_[_] is PCxt P which additionally requires that all terms contained in the evaluation context (that is one or zero terms) satisfy predicate P. This allows us the formulation of P-neutrals as terms of the form $\vec{E}[x]$ for some $\vec{E}[\_] = E_1[\ldots E_n[\_]]$ and a variable x where all immediate subterms satisfy P.

```
data PCxt  {Γ} (P : ∀{c} → Tm Γ c → Set) :
           ∀ {a b} → Tm Γ b → ECxt Γ a b → Tm Γ a → Set where
  appl  :  ∀ {a b t u}     (u : P u)  → PCxt P (app t u)  (appl u)  (t : (a ⇀̂ b))
  fst   :  ∀ {a b t}                  → PCxt P (fst t)     fst      (t : (a ×̂ b))
  snd   :  ∀ {a b t}                  → PCxt P (snd t)     snd      (t : (a ×̂ b))
  *l_   :  ∀ {a∞ b∞ t u}   (u : P u)  → PCxt P (t * (u : ▶̂ a∞) : ▶̂ b∞) (*l u) t
  *r_   :  ∀ {a∞ b∞ t u}   (t : P (next {a∞ = a∞ ⇒ b∞} t))
```

```
                                  → PCxt P ((next t) ∗ (u : ▸̂ a∞) : ▸̂ b∞) (∗r t) u

data PNe  {Γ} (P : ∀{c} → Tm Γ c → Set) {b} : Tm Γ b → Set where
  var   :  ∀  x                                  → PNe P (var x)
  elim  :  ∀  {a} {t : Tm Γ a} {E Et}
           → (n : PNe P t) (Et : PCxt P Et E t)  → PNe P Et
```

Weak head reduction (whr) is a reduction of the form $\vec{E}[t] \longrightarrow \vec{E}[t']$ where $t \mapsto t'$. It is well-known that weak head expansion (whe) does not preserve sn, e.g., $(\lambda x.\, y)\Omega$ is not sn even though it contracts to y. In this case, Ω is a *vanishing term* lost by reduction. If we require that all vanishing terms in a reduction are sn, weak head expansion preserves sn. In the following, we define P-whr where all vanishing terms must satisfy P.

```
data _/_⇒_  {Γ} (P : ∀{c} → Tm Γ c → Set) :
            ∀ {a} → Tm Γ a → Tm Γ a → Set where

  β     : ∀ {a b}{t : Tm (a :: Γ) b}{u}
          → (u : P u)
          → P / (app (abs t) u) ⇒ subst0 u t

  βfst  : ∀ {a b}{t : Tm Γ a}{u : Tm Γ b}
          → (u : P u)
          → P / fst (pair t u) ⇒ t

  βsnd  : ∀ {a b}{t : Tm Γ a}{u : Tm Γ b}
          → (t : P t)
          → P / snd (pair t u) ⇒ u

  β▸    : ∀ {a∞ b∞}{t : Tm Γ (force (a∞ ⇒ b∞))}{u : Tm Γ (force a∞)}
          → P / (next t ∗ next {a∞ = a∞} u) ⇒ (next {a∞ = b∞} (app t u))

  cong  : ∀ {a b t t' Et Et'}{E : ECxt Γ a b}
          → (Et  : Et ≅ E [ t ])
          → (Et' : Et' ≅ E [ t' ])
          → (t⇒  : P / t ⇒ t')
          → P / Et ⇒ Et'
```

The family of predicates $\mathsf{SN}\ n$ is defined inductively by the following rules—we allow ourselves set-notation at this semi-formal level:

$$\frac{t \in \mathsf{SN}\ n}{\lambda x t \in \mathsf{SN}\ n} \qquad \frac{t_1, t_2 \in \mathsf{SN}\ n}{(t_1, t_2) \in \mathsf{SN}\ n} \qquad \frac{}{\mathsf{next}\ t \in \mathsf{SN}\ 0} \qquad \frac{t \in \mathsf{SN}\ n}{\mathsf{next}\ t \in \mathsf{SN}\ (1+n)}$$

$$\frac{t \in \mathsf{SNe}\ n}{t \in \mathsf{SN}\ n} \qquad \frac{t' \in \mathsf{SN}\ n \qquad t\ \langle n \rangle\!\Rightarrow t'}{t \in \mathsf{SN}\ n}$$

The last two rules close SN under neutrals SNe, which is an instance of PNe with $P = \mathsf{SN}\ n$, and level-n *strong head expansion* $t\ \langle n \rangle\!\Rightarrow t'$, which is an instance of P-whe with also $P = \mathsf{SN}\ n$. We represent the inductive SN in Agda as a sized type (Hughes et al., 1996; Abel and Pientka, 2013) for the purpose of termination checking certain inductions on SN later. The assignment of sizes follows the principle that recursive invokations of SN within a constructor of $\mathsf{SN}\ \{i\}$ must carry a strictly smaller size $j : \mathsf{Size}< i$. The mutually defined relations $\mathsf{SNe}\ n\ t$ (instance of PNe) and strong head reduction (shr) $t\ \langle n \rangle\!\Rightarrow t'$ just thread the size argument through. Note that there is a version $i\ \mathsf{size}\ t\ \langle n \rangle\!\Rightarrow t'$ of shr that makes the size argument visible, to be supplied in case exp.

```
mutual
  data SN {i : Size}{Γ} : (n : ℕ) → ∀ {a} → Tm Γ a → Set where

    abs    : ∀ {j : Size< i} {a b n}{t : Tm (a :: Γ) b}
             → (t : SN {j} n t)
             → SN n (abs t)

    pair   : ∀ {j₁ j₂ : Size< i} {a b n t u}
             → (t : SN {j₁} n t) (u : SN {j₂} n u)
             → SN n {a ^× b} (pair t u)

    next0  : ∀ {a∞} {t : Tm Γ (force a∞)}
             → SN 0 {▶ a∞} (next t)

    next   : ∀ {j : Size< i} {a∞ n} {t : Tm Γ (force a∞)}
             → (t : SN {j} n t)
             → SN (1 + n) {▶^ a∞} (next t)

    ne     : ∀ {j : Size< i} {a n t}
             → (n : SNe {j} n t)
             → SN n {a} t

    exp    : ∀ {j₁ j₂ : Size< i} {a n t t'}
             → (t⇒ : j₁ size t ⟨ n ⟩⇒ t') (t' : SN {j₂} n t')
             → SN n {a} t

  SNe              : ∀ {i : Size} {Γ a} (n : ℕ) → Tm Γ a → Set
  SNe {i} n = PNe (SN {i} n)

  _size_⟨_⟩⇒_ : ∀ (i : Size) {Γ a} → Tm Γ a → ℕ → Tm Γ a → Set
  i size t ⟨ n ⟩⇒ t' = SN {i} n / t ⇒ t'

  _⟨_⟩⇒_           : ∀ {i : Size} {Γ a} → Tm Γ a → ℕ → Tm Γ a → Set
  _ ⟨ _ ⟩⇒ _ {i} t n t' = SN {i} n / t ⇒ t'
```

The SN-relations are antitone in the level n. This is one dimension of the Kripke worlds in our model (see next section).

```
mapSN  : ∀ {m n} → m ≤ℕ n → ∀ {Γ a}{t  : Tm Γ a} → SN n t  → SN m t


mapSNe : ∀ {m n} → m ≤ℕ n → ∀ {Γ a}{t  : Tm Γ a} → SNe n t  → SNe m t
map⇒   : ∀ {m n} → m ≤ℕ n → ∀ {Γ a}{t t' : Tm Γ a} → t ⟨ n ⟩⇒ t' → t ⟨ m ⟩⇒ t'
```

The other dimension of the Kripke worlds is the typing context; our notions are also closed under renaming (and even undoing of renaming). Besides renameSN, we have analogous lemmata renameSNe and rename⇒.

```
renameSN  : ∀ {n a Δ Γ} (ρ : Δ ≤ Γ) {t : Tm Γ a} →
            SN n t → SN n (rename ρ t)

fromRenameSN : ∀{n a Γ Δ} (ρ : Δ ≤ Γ) {t : Tm Γ a} →
               SN n (rename ρ t) → SN n t
```

A consequence of fromRenameSN is that $t \in \mathsf{SN}\ n$ iff $t\ x \in \mathsf{SN}\ n$ for some variable x. (Consider $t = \lambda y.t'$ and $t\ x\ \langle n \rangle\Rightarrow t'[y/x]$.) This property is essential for the construction of the function space on sn sets (see next section).

```
absVarSN  : ∀{Γ a b n}{t : Tm (a :: Γ) (a ↝ b)} →
            app t (var zero) ∈ SN n → t ∈ SN n
```

6 Soundness

A well-established technique (Tait, 1967) to prove strong normalization is to model each type a as a set $\mathscr{A} = [\![a]\!]$ of sn terms. Each so-called semantic type $\mathscr{A}$ should contain the variables in order to interpret open terms by themselves (using the identity valuation). To establish the conditions of semantic types compositionally, the set $\mathscr{A}$ needs to be *saturated*, i. e., contain SNe (rather than just the variables) and be closed under strong head expansion (to entertain introductions).

As a preliminary step towards saturated sets we define sets of well-typed terms in an arbitrary typing context but fixed type, TmSet a. We also define shorthands for the largest set, set inclusion and closure under expansion.

```
TmSet : (a : Ty) → Set₁
TmSet a = {Γ : Cxt} (t : Tm Γ a) → Set

[⊤] : ∀{a} → TmSet a
[⊤] t = ⊤

_⊆_ : ∀{a} (A A′ : TmSet a) → Set
A ⊆ A′ = ∀{Γ}{t : Tm Γ _} → A t → A′ t

Closed : ∀ (n : ℕ) {a} (A : TmSet a) → Set
Closed n A = ∀{Γ}{t t′ : Tm Γ _} → t ⟨ n ⟩⇒ t′ → A t′ → A t
```

For each type constructor we define a corresponding operation on TmSets. The product is simply pointwise through the use of the projections.

```
_[×]_ : ∀{a b} → TmSet a → TmSet b → TmSet (a ×̂ b)
(𝒜 [×] ℬ) t = 𝒜 (fst t) × ℬ (snd t)
```

For function types we are forced to use a Kripke-style definition, quantifying over all possible extended contexts Δ makes $\mathscr{A}$ [→] $\mathscr{B}$ closed under renamings.

```
_[→]_ : ∀{a b} → TmSet a → TmSet b → TmSet (a →̂ b)
(𝒜 [→] ℬ) {Γ} t = ∀{Δ} (ρ : Δ ≤ Γ) → ∀ {u} → 𝒜 u → ℬ (app (rename ρ t) u)
```

The TmSet for the later modality is indexed by the depth. The first two constructors are for terms in the canonical form next t, at depth zero we impose no restriction on t, otherwise we use the given set A. The other two constructors are needed to satisfy the properties we require of our saturated sets.

```
data [▶] {a∞} (A : TmSet (force a∞)) {Γ} : (n : ℕ) → Tm Γ (▶̂ a∞) → Set where
  next0 : ∀ {t : Tm Γ (force a∞)}                  → [▶] A zero    (next t)
  next  : ∀ {n}{t : Tm Γ (force a∞)} (t : A t)     → [▶] A (suc n) (next t)
  ne    : ∀ {n}{t : Tm Γ (▶̂ a∞)}     (n : SNe n t) → [▶] A n       t
  exp   : ∀ {n}{t t′ : Tm Γ (▶̂ a∞)}
          (t⇒ : t ⟨ n ⟩⇒ t′)          (t : [▶] A n t′) → [▶] A n       t
```

The particularity of our saturated sets is that they are indexed by the depth, which in our case is needed to state the usual properties. In particular if a term belongs to a saturated set it is also a member of SN, which is what we need for strong normalization. In addition we require them to be closed under renaming, since we are dealing with terms in a context.

```
record IsSAT (n : ℕ) {a} (A : TmSet a) : Set where
  field
```

```
    satSNe      : SNe n ⊆ A
    satSN       : A     ⊆ SN n
    satExp      : Closed n A
    satRename   : ∀ {Γ Δ} (ρ : Δ ≤ Γ) → ∀ {t} → A t → A (rename ρ t)

record SAT (a : Ty) (n : ℕ) : Set₁ where
  field
    satSet  : TmSet a
    satProp : IsSAT n satSet
```

For function types we will also need a notion of a sequence of saturated sets up to a specified maximum depth n.

```
SAT≤ : (a : Ty) (n : ℕ) → Set₁
SAT≤ a n = ∀ {m} → m ≤ℕ n → SAT a m
```

To help Agda's type inference, we also define a record type for membership of a term into a saturated set.

```
record _∈_ {a n Γ} (t : Tm Γ a) (𝒜 : SAT a n) : Set where
  constructor ↑_
  field ↓_ : satSet 𝒜 t

_∈⟨_⟩_ : ∀ {a n Γ} (t : Tm Γ a) {m} (m≤n : m ≤ℕ n) (𝒜 : SAT≤ a n) → Set
t ∈⟨ m≤n ⟩ 𝒜 = t ∈ 𝒜 m≤n
```

Given the lemmas about SN shown so far we can lift our operations on TmSet to saturated sets and give the semantic version of our term constructors.

For function types we need another level of Kripke-style generalization to smaller depths, so that we can maintain antitonicity.

```
_⟦→⟧_ : ∀ {n a b} (𝒜 : SAT≤ a n) (ℬ : SAT≤ b n) → SAT (a ⇒ b) n
𝒜 ⟦→⟧ ℬ = record
  { satSet  = λ t → ∀ m (m≤n : m ≤ℕ _) → (A m≤n [→] B m≤n) t
  ; satProp = record
    { satSN  = CSN
  ; satSNe    = CSNe
  ; satExp    = CExp
  ; satRename = CRename
    }
  }
  where
    module 𝒜 = SAT≤ 𝒜
    module ℬ = SAT≤ ℬ
    A = 𝒜.satSet
    B = ℬ.satSet

    C : TmSet (_ ⇒ _)
    C t = ∀ m (m≤n : m ≤ℕ _) → (A m≤n [→] B m≤n) t

    CSN : C ⊆ SN _
    CSN t = fromRenameSN suc (absVarSN
      (ℬ.satSN ≤ℕ.refl (t _ ≤ℕ.refl suc (𝒜.satSNe ≤ℕ.refl (var zero)))))
  CSNe : SNe _ ⊆ C
  CSNe n m m≤n ρ u =
    ℬ.satSNe m≤n (sneApp (mapSNe m≤n (renameSNe ρ n)) (𝒜.satSN m≤n u))

  CExp : ∀{Γ}{t t' : Tm Γ _} → t ⟨ _ ⟩⇒ t' → C t' → C t
  CExp t⇒ t m m≤n ρ u =
    ℬ.satExp m≤n ((cong (appl _) (appl _) (map⇒ m≤n (rename⇒ ρ t⇒)))) (t m m≤n ρ u)

  CRename : {Γ Δ : List Ty} (ρ : Δ ≤ Γ) {t : Tm Γ _} → C t → C (rename ρ t)
```

```
CRename = λ ρ {t} t m m≤n ρ' {u} u →
  ≡.subst (λ t₁ → B {m} m≤n (app t₁ u)) (subst-• ρ' ρ t) (t m m≤n (ρ' •s ρ) u)
```

The proof of inclusion into SN first derives that app (rename suc t) (var zero) is in SN through the inclusion of neutral terms into $\mathscr{A}$ and the inclusion of $\mathscr{B}$ into SN, then proceeds to strip away first (var zero) and then (rename suc), so that we are left with the original goal SN n t. Renaming t with suc is necessary to be able to introduce the fresh variable zero of type a.

The types of semantic abstraction and application are somewhat obfuscated because they need to mention the upper bounds and the renamings.

```
⟦abs⟧ : ∀ {n a b} {𝒜 : SAT≤ a n} {ℬ : SAT≤ b n} {Γ} {t : Tm (a :: Γ) b} →
          (∀ {m} (m≤n : m ≤ℕ n) {Δ} (ρ : Δ ≤ Γ) {u : Tm Δ a} →
             u ∈⟨ m≤n ⟩ 𝒜 → (subst0 u (subst (lifts ρ) t)) ∈⟨ m≤n ⟩ ℬ)
        → abs t ∈ (𝒜 ⟦→⟧ ℬ)
(↓ ⟦abs⟧ {𝒜 = 𝒜}{ℬ = ℬ} t) m m≤n ρ u =
  SAT≤.satExp ℬ m≤n (β (SAT≤.satSN 𝒜 m≤n u)) (↓ t m≤n ρ (↑ u))

⟦app⟧ : ∀ {n a b}{𝒜 : SAT≤ a n}{ℬ : SAT≤ b n}{Γ}{t : Tm Γ (a →̂ b)}{u : Tm Γ a}
        → t ∈ (𝒜 ⟦→⟧ ℬ) → u ∈⟨ ≤ℕ.refl ⟩ 𝒜 → app t u ∈⟨ ≤ℕ.refl ⟩ ℬ
⟦app⟧ {ℬ = ℬ} {u = u} (↑ t) (↑ u) = ≡.subst (λ t → app t u ∈⟨ ≤ℕ.refl ⟩ ℬ) renId
                                      (↑ t _ ≤ℕ.refl id u)
```

The TmSet for product types is directly saturated, inclusion into SN uses a lemma to derive SN n t from SN n (fst t), which follows from $\mathscr{A} \subseteq$ SN.

```
_⟦×⟧_ : ∀ {n a b} (𝒜 : SAT a n) (ℬ : SAT b n) → SAT (a ×̂ b) n
𝒜 ⟦×⟧ ℬ = record
  { satSet  = satSet 𝒜 [×] satSet ℬ
  ; satProp = record
    { satSNe     = CSNe
    ; satSN      = CSN
    ; satExp     = CExp
    ; satRename = λ ρ x → satRename 𝒜 ρ (proj₁ x) , satRename ℬ ρ (proj₂ x)
    }
  }
  where
    A = satSet 𝒜
    B = satSet ℬ
    C : TmSet _
    C = A [×] B

    CSNe          : SNe _ ⊆ C
    CSNe n        = satSNe 𝒜 (elim n fst)
                  , satSNe ℬ (elim n snd)

    CSN           : C ⊆ SN
    CSN (t , u)   = bothProj SN (satSN 𝒜 t) (satSN ℬ u)

    CExp          : ∀{Γ}{t t' : Tm Γ _} → t ⟨ _ ⟩⇒ t' → C t' → C t
    CExp t⇒ (t , u) = satExp 𝒜 (cong fst fst t⇒) t
                    , satExp ℬ (cong snd snd t⇒) u
```

Semantic introduction ⟦pair⟧ : $t_1 \in \mathscr{A} \to t_2 \in \mathscr{B} \to$ pair t_1 $t_2 \in (\mathscr{A}$ ⟦×⟧ $\mathscr{B})$ and eliminations ⟦fst⟧ : $t \in (\mathscr{A}$ ⟦×⟧ $\mathscr{B}) \to$ fst $t \in \mathscr{A}$ and ⟦snd⟧ : $t \in (\mathscr{A}$ ⟦×⟧ $\mathscr{B}) \to$ snd $t \in \mathscr{B}$ for pairs are straightforward.

```
⟦pair⟧ : ∀ {n a b} {𝒜 : SAT a n} {ℬ : SAT b n} {Γ} {t₁ : Tm Γ a} {t₂ : Tm Γ b}
         → t₁ ∈ 𝒜 → t₂ ∈ ℬ → pair t₁ t₂ ∈ (𝒜 ⟦×⟧ ℬ)
↓ ⟦pair⟧ {𝒜 = 𝒜} {ℬ = ℬ} (↑ t) (↑ u) = satExp 𝒜 (βfst (satSN ℬ u)) t
                                     , satExp ℬ (βsnd (satSN 𝒜 t)) u
```

```
⟦fst⟧   :  ∀ {n a b} {𝒜 : SAT a n} {ℬ : SAT b n} {Γ} {t : Tm Γ (a ×̂ b)}
           → t ∈ (𝒜 ⟦×⟧ ℬ) → fst t ∈ 𝒜
⟦fst⟧ t = 1 (proj₁ (↓ t))

⟦snd⟧   :  ∀ {n a b} {𝒜 : SAT a n} {ℬ : SAT b n} {Γ} {t : Tm Γ (a ×̂ b)}
           → t ∈ (𝒜 ⟦×⟧ ℬ) → snd t ∈ ℬ
⟦snd⟧ t = 1 (proj₂ (↓ t))
```

The later modality is going to use the saturated set for its type argument at the preceeding depth, we encode this fact through the type SATpred.

```
SATpred : (a : Ty) (n : ℕ) → Set₁
SATpred a zero    = ⊤
SATpred a (suc n) = SAT a n

SATpredSet : {n : ℕ}{a : Ty} → SATpred a n → TmSet a
SATpredSet {zero}  𝒜 = [⊤]
SATpredSet {suc n} 𝒜 = satSet 𝒜
```

Since the cases for ⟦▶⟧_ are essentially a subset of those for SN, the proof of inclusion into SN goes by induction and the inclusion of 𝒜 into SN.

```
⟦▶⟧_ : ∀{n a∞} (𝒜 : SATpred (force a∞) n) → SAT (▶̂ a∞) n
⟦▶⟧_ {n} {a∞} 𝒜 = record
  { satSet = ⟦▶⟧ (SATpredSet 𝒜) n
  ; satProp = record
    { satSNe    = ne
    ; satSN     = CSN 𝒜
    ; satExp    = exp
    ; satRename = CRen 𝒜
    }
  }
  where
    𝑪 : ∀ {n} (𝒜 : SATpred (force a∞) n) → TmSet (▶̂ a∞)
    𝑪 {n} 𝒜 = ⟦▶⟧ (SATpredSet 𝒜) n

    CSN : ∀ {n} (𝒜 : SATpred (force a∞) n) → 𝑪 {n} 𝒜 ⊆ SN n
    CSN 𝒜 next0      = next0
    CSN 𝒜 (next t)   = next (satSN 𝒜 t)
    CSN 𝒜 (ne n)     = ne n
    CSN 𝒜 (exp t⇒ t) = exp t⇒ (CSN 𝒜 t)

    CRen : ∀ {n} (𝒜 : SATpred (force a∞) n) → ∀ {Γ Δ} (ρ : Γ ≤ Δ) →
           ∀ {t} → 𝑪 {n} 𝒜 t → 𝑪 {n} 𝒜 (subst ρ t)
    CRen 𝒜 ρ next0      = next0
    CRen 𝒜 ρ (next t)   = next (satRename 𝒜 ρ t)
    CRen 𝒜 ρ (ne n)     = ne (renameSNe ρ n)
    CRen 𝒜 ρ (exp t⇒ t) = exp (rename⇒ ρ t⇒) (CRen 𝒜 ρ t)
```

Following Section 3 we can assemble the combinators for saturated sets into a semantics for the types of $\lambda^{\blacktriangleright}$. The definition of ⟦_⟧_ proceeds by recursion on the inductive part of the type, and otherwise by well-founded recursion on the depth. Crucially the interpretation of the later modality only needs the interpretation of its type parameter at a smaller depth, which is then decreasing exactly when the representation of types becomes coinductive and would no longer support recursion.

```
⟦_⟧≤ : (a : Ty) {n : ℕ} → ∀ {m} → m ≤ℕ n → SAT a m

⟦_⟧_ : (a : Ty) (n : ℕ) → SAT a n
⟦ a →̂ b ⟧ n = ⟦ a ⟧≤ {n} ⟦→⟧ ⟦ b ⟧≤ {n}
⟦ a ×̂ b ⟧ n = ⟦ a ⟧ n ⟦×⟧ ⟦ b ⟧ n
```

```
⟦ ▶̂ a∞ ⟧ n = ⟦▶⟧ P n
  where
    P : ∀ n → SATpred (force a∞) n
    P zero    =
    P (suc n) = ⟦ force a∞ ⟧ n
```

Well-founded recursion on the depth is accomplished through the auxiliary definition $[\![\_]\!]{\leq}$ which recurses on the inequality proof. It is however straightforward to convert in and out of the original interpretation, or between different upper bounds.

```
in≤  : ∀ a {n m} (m≤n : m ≤ℕ n) → satSet (⟦ a ⟧  m)   ⊆ satSet (⟦ a ⟧≤ m≤n)
out≤ : ∀ a {n m} (m≤n : m ≤ℕ n) → satSet (⟦ a ⟧≤ m≤n) ⊆ satSet (⟦ a ⟧ m)

coerce≤ : ∀ a {n n' m} (m≤n : m ≤ℕ n) (m≤n' : m ≤ℕ n')
        → satSet (⟦ a ⟧≤ m≤n) ⊆ satSet (⟦ a ⟧≤ m≤n')
```

As will be necessary later for the interpretation of next, the interpretation of types is also antitone. For most types this follows by recursion, while for function types antitonicity is embedded in their semantics and we only need to convert between different upper bounds.

```
map⟦_⟧ : ∀ a {m n} → m ≤ℕ n → satSet (⟦ a ⟧ n) ⊆ satSet (⟦ a ⟧ m)
```

```
map⟦ a →̂ b ⟧ m≤n t          = λ l l≤m ρ u → let l≤n = ≤ℕ.trans l≤m m≤n in
                              coerce≤ b l≤n l≤m (t l l≤n ρ (coerce≤ a l≤m l≤n u))
map⟦ a ×̂ b ⟧ m≤n (t , u)    = map⟦ a ⟧ m≤n t , map⟦ b ⟧ m≤n u
map⟦ ▶ a∞ ⟧ m≤n (ne n)      = ne (mapSNe m≤n n)
map⟦ ▶̂ a∞ ⟧ m≤n (exp t⇒ t)  = exp (map⇒ m≤n t⇒) (map⟦ ▶̂ a∞ ⟧ m≤n t)
map⟦ ▶ a∞ ⟧ {m = zero} m≤n next0      = next0
map⟦ ▶̂ a∞ ⟧ {m = suc m} () next0
map⟦ ▶ a∞ ⟧ {m = zero} m≤n (next _)   = next0
map⟦ ▶̂ a∞ ⟧ {m = suc m} m≤n (next t)  = next (map⟦ force a∞ ⟧ (pred≤ℕ m≤n) t)
```

Typing contexts are interpreted as predicates on substitutions. These predicates inherit antitonicity and closure under renaming. Semantically sound substitutions act as environments θ. We will need Ext to extend the environment for the interpretation of lambda abstractions.

```
⟦_⟧C : ∀ Γ {n} → ∀ {Δ} (σ : Subst Γ Δ) → Set
⟦ Γ ⟧C {n} σ = ∀ {a} (x : Var Γ a) → σ x ∈ ⟦ a ⟧ n

Map : ∀ {m n} → (m≤n : m ≤ℕ n) →
      ∀ {Γ Δ} {σ : Subst Γ Δ} (θ : ⟦ Γ ⟧C {n} σ) → ⟦ Γ ⟧C {m} σ
Map m≤n θ {a} x = map⟦ a ⟧∈ m≤n (θ x)

Rename : ∀ {n Δ Δ'} → (ρ : Ren Δ Δ') →
         ∀ {Γ}{σ : Subst Γ Δ} (θ : ⟦ Γ ⟧C {n} σ) →
         ⟦ Γ ⟧C (ρ •s σ)
Rename ρ θ {a} x = ↑ satRename (⟦ a ⟧ _) ρ (↓ θ x)

Ext : ∀ {a n Δ Γ} {t : Tm Δ a} → (t : t ∈ ⟦ a ⟧ n) →
      ∀ {σ : Subst Γ Δ} (θ : ⟦ Γ ⟧C σ) → ⟦ a :: Γ ⟧C (t ::s σ)
Ext t θ (zero) = t
Ext t θ (suc x) = θ x
```

The soundness proof, showing that every term of $\lambda^{\blacktriangleright}$ is a member of our saturated sets and so a member of SN, is now a simple matter of interpreting each operation in the language to its equivalent in the semantics that we have defined so far.

```
sound : ∀ {n a Γ} (t : Tm Γ a) {Δ} {σ : Subst Γ Δ} →
        (θ : ⟦ Γ ⟧C {n} σ) → subst σ t ∈ ⟦ a ⟧ n
sound (var x) θ = θ x
sound (abs t) θ = ⟦abs⟧ {t = t} λ m≤n ρ u →
  ↑ in≤ _ m≤n (↓ sound t (Ext (↑ out≤ _ m≤n (↓ u)) (Rename ρ (Map m≤n θ))))
sound (app t u)  θ = ⟦app⟧  (sound t θ)  (sound u θ)
sound (pair t u) θ = ⟦pair⟧ (sound t θ)  (sound u θ)
sound (fst t)    θ = ⟦fst⟧  (sound t θ)
sound (snd t)    θ = ⟦snd⟧  (sound t θ)
sound (t * u)    θ = ⟦*⟧    (sound t θ)  (sound u θ)
sound {zero} (next t) θ = ↑ next0
sound {suc n} (next t) θ = ↑ (next (↓ sound t (Map n≤sn θ)))
```

The interpretation of next depends on the depth, at zero we are done, at suc n we recurse on the subterm at depth n, using antitonicity to Map the current environment to depth n as well. In fact without next we would not have needed antitonicity at all since there would have been no way to embed a term from a smaller depth into a larger one.

7 SN correctness

To complete our strong normalization proof we need to show that SN is included in the characterization of strong normalization as a well-founded predicate sn.

```
fromSN : ∀ {i} {Γ} {n : ℕ} {a} {t : Tm Γ a} →
         SN {i} n t → sn n t
```

The cases for canonical and neutral forms are straightforward, since no reduction can happen at the top of the expression and we cover the others through the induction hypotheses.

```
fromSNe : ∀ {i Γ n a} {t : Tm Γ a} →
          SNe {i} n t → sn n t

fromSN (ne n)       = fromSNe n
fromSN (abs t)      = abssn (fromSN t)
fromSN (pair t u)   = pairsn (fromSN t) (fromSN u)
fromSN next0        = next0sn
fromSN (next t)     = nextsn (fromSN t)
fromSN (exp t⇒ t₁)  = acc (expsn t⇒ t₁ (fromSN t₁))
```

The expansion case is more challenging instead, we can not in fact prove expsn by induction directly.

```
expsn : ∀ {i j Γ n a} {t th to : Tm Γ a} →
        i size t ⟨ n ⟩⇒ th → SN {j} n th → sn n th →
        t ⟨ n ⟩⇒β to → sn n to
```

We can see the problem by looking at one of the congruence cases, in particular reduction on the left of an application. There we would have $t\,u \in sn$, $t h t_1$ and $t \beta t_2$, and need to prove $t_2\,u \in sn$. By induction we could obtain $t_2 \in sn$ but then there would be no easy way to obtain $t_2\,u \in sn$, since strong normalization is not closed under application.

The solution is to instead generalize the statement to work under a sequence of head reduction evaluation contexts. We represent such sequences with the type ECxt*, and denote their application to a term with the operator _[_]*.

```
expsnCxt : ∀ {i j Γ n a b} {t th to : Tm Γ a} →
           (Es : ECxt* Γ a b) → i size t ⟨ n ⟩⇒ th →
           SN {j} n (Es [ th ]*) → sn n (Es [ th ]*) →
           t ⟨ n ⟩⇒β to → sn n (Es [ to ]*)
expsn t⇒ t t⇒β = expsnCxt [] t⇒ t t⇒β
```

In this way the congruence cases are solved just by induction with a larger context.

```
expsnCxt E (cong (appl u) (appl .u) th⇒) th th (cong (appl .u) (appl .u) t⇒)
  = expsnCxt (appl u :: E) th⇒ th th t⇒
```

This generalization however affects the lemmata that handle the reduction cases, which also need to work under a sequence of evaluation contexts. Fortunately the addition of a premise $E[z] \in sn$, about an unrelated term z, allows to conveniently handle all the reductions that target the context.

```
β▶sn : ∀ {n Γ b} {a∞ b∞} {z} {t : Tm Γ (force (a∞ ⇒ b∞))} {u : Tm Γ (force a∞)}
       (E : ECxt* Γ (▶ b∞) b) → sn (suc n) (E [ z ]*) →
       sn n t → sn n u → sn (suc n) (E [ next (app t u) ]*) →
       sn (suc n) (E [ next t * next {a∞ = a∞} u ]*)


βfstsn : ∀ {n Γ b} {a c} {z} {t : Tm Γ b} {u : Tm Γ a}
         (E : ECxt* Γ b c) → sn n (E [ z ]*) →
         sn n t → sn n u → sn n (E [ t ]*) →
         sn n (E [ fst (pair t u) ]*)


βsndsn : ∀ {n Γ b} {a c} {z} {t : Tm Γ b} {u : Tm Γ a}
         (E : ECxt* Γ b c) → sn n (E [ z ]*) →
         sn n t → sn n u → sn n (E [ t ]*) →
         sn n (E [ snd (pair u t) ]*)


βsn : ∀ {i n a b c Γ} {u : Tm Γ a} {t : Tm (a :: Γ) b}{z}
      (Es : ECxt* Γ b c) → sn n (Es [ z ]*) →
      sn n t → SN {i} n (Es [ subst0 u t ]*) → sn n u →
      sn n (Es [ app (abs t) u ]*)
```

8 Conclusions

In this paper, we presented a family of strongly-normalizing reduction relations for simply-typed lambda calculus with Nakano's modality for recursion. Using a similar stratification, Krishnaswami and Benton (2011a) have shown weak normalization using hereditary substitutions, albeit for a system without recursive types.

Our Agda formalization uses a saturated sets semantics based on an inductive notion of strong normalization. Herein, we represented recursive types as infinite type expressions and terms as intrinsically well-typed ones.

Our treatment of infinite type expressions was greatly simplified by adding an extensionality axiom for the underlying coinductive type to Agda's type theory. This would

not have been necessary in a more extensional theory such as *Observational Type Theory* (Altenkirch et al., 2007) as shown in (McBride, 2009). Possibly *Homotopy Type Theory* (UnivalentFoundations, 2013) would also address this problem, but there the status of coinductive types is yet unclear.

For the future, we would like to investigate how to incorporate guarded recursive types into a dependently-typed language, and how they relate to other approaches like coinduction with sized types, for instance.

Acknowledgments. Thanks to Lars Birkedal, Ranald Clouston, and Rasmus Møgelberg for fruitful discussions on guarded recursive types, and Hans Bugge Grathwohl, Fabien Renaud, and some anonymous referees for useful feedback on the Agda development and a draft version of this paper. The first author acknowledges support by Vetenskapsrådet framework grant 254820104 (Thierry Coquand). This paper has been prepared with Stevan Andjelkovic's Agda-to-LaTeX converter.

References

Agda Wiki. Chalmers and Gothenburg University, 2.4 edn. (2014), http://wiki.portal.chalmers.se/agda

Abel, A.: Normalization for the simply-typed lambda-calculus in Twelf. In: Logical Frameworks and Metalanguages (LFM 04). Electronic Notes in Theoretical Computer Science, vol. 199C, pp. 3–16. Elsevier (2008)

Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Proc. of the 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'13. pp. 185–196. ACM Press (2013)

Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013. pp. 27–38. ACM Press (2013)

Abel, A., Vezzosi, A.: A formalized proof of strong normalization for guarded recursive types (long version and Agda sources) (Aug 2014), http://www.cse.chalmers.se/~abela/publications.html#aplas14

Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now! In: Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007. pp. 57–68. ACM Press (2007)

Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1683, pp. 453–468. Springer-Verlag (1999)

Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: Proc. of the 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'13. pp. 197–208. ACM Press (2013)

Benton, N., Hur, C.K., Kennedy, A., McBride, C.: Strongly typed term representations in Coq. Journal of Automated Reasoning 49(2), 141–159 (2012)

Birkedal, L., Møgelberg, R.E.: Intensional type theory with guarded recursive types qua fixed points on universes. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013. pp. 213–222. IEEE Computer Society Press (2013)

Dybjer, P.: Inductive families. Formal Aspects of Computing 6(4), 440–465 (1994)

Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996. pp. 410–423 (1996)

Joachimski, F., Matthes, R.: Short proofs of normalization. Archive of Mathematical Logic 42(1), 59–87 (2003)

Krishnaswami, N.R., Benton, N.: A semantic model for graphical user interfaces. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 45–57. ACM Press (2011a)

Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada. pp. 257–266. IEEE Computer Society Press (2011b)

McBride, C.: Type-preserving renaming and substitution (2006), http://strictlypositive.org/ren-sub.pdf , unpublished draft

McBride, C.: Let's see how things unfold: Reconciling the infinite with the intensional. In: Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5728, pp. 113–126. Springer-Verlag (2009)

McBride, C.: Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In: Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010. pp. 1–12. ACM Press (2010)

Nakano, H.: A modality for recursion. In: 15th Annual IEEE Symposium on Logic in Computer Science (LICS 2000), 26-29 June 2000, Santa Barbara, California, USA, Proceedings. pp. 255–266. IEEE Computer Society Press (2000)

van Raamsdonk, F., Severi, P., Sørensen, M.H., Xi, H.: Perpetual reductions in lambda calculus. Information and Computation 149(2), 173–225 (1999)

Tait, W.W.: Intensional interpretations of functionals of finite type I. The Journal of Symbolic Logic 32(2), 198–212 (1967)

UnivalentFoundations: Homotopy type theory: Univalent foundations of mathematics. Tech. rep., Institute for Advanced Study (2013), http://homotopytypetheory.org/book/

Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 224–235. New Orleans (2003)

Chapter 4

Guarded Cubical Type Theory: Path Equality for Guarded Recursion

Guarded Cubical Type Theory: Path Equality for Guarded Recursion

Lars Birkedal[1], Aleš Bizjak[1], Ranald Clouston[1], Hans Bugge Grathwohl[1], Bas Spitters[1], and Andrea Vezzosi[2]

[1]Department of Computer Science, Aarhus University, Denmark
[2]Department of Computer Science and Engineering, Chalmers University of Technology, Sweden

Abstract

This paper improves the treatment of equality in guarded dependent type theory (GDTT), by combining it with cubical type theory (CTT). GDTT is an extensional type theory with guarded recursive types, which are useful for building models of program logics, and for programming and reasoning with coinductive types. We wish to implement GDTT with decidable type checking, while still supporting non-trivial equality proofs that reason about the extensions of guarded recursive constructions. CTT is a variation of Martin-Löf type theory in which the identity type is replaced by abstract paths between terms. CTT provides a computational interpretation of functional extensionality, is conjectured to have decidable type checking, and has an implemented type checker. Our new type theory, called guarded cubical type theory, provides a computational interpretation of extensionality for guarded recursive types. This further expands the foundations of CTT as a basis for formalisation in mathematics and computer science. We present examples to demonstrate the expressivity of our type theory, all of which have been checked using a prototype type-checker implementation, and present semantics in a presheaf category.

1 Introduction

Guarded recursion is a technique for defining and reasoning about infinite objects. Its applications include the definition of productive operations on data structures more commonly defined via coinduction, such as streams, and the construction of models of program logics for modern programming languages with features such as higher-order store and concurrency [6]. This is done via the type-former $\triangleright$, called 'later', which distinguishes data which is available immediately from data only available after some computation, such as the unfolding of a fixed-point. For example, guarded recursive streams are defined by the equation

$$\mathsf{Str}_A \;=\; A \times \triangleright \mathsf{Str}_A$$

rather than the more standard $\mathsf{Str}_A = A \times \mathsf{Str}_A$, to specify that the head is available now but the tail only later. The type for fixed-point combinators is then $(\triangleright A \to A) \to A$, rather than the logically inconsistent $(A \to A) \to A$, disallowing unproductive definitions such as taking the fixed-point of the identity function.

Guarded recursive types were developed in a simply-typed setting by Clouston et al. [9], following earlier work [21, 3, 1], alongside a logic for reasoning about such programs. For large examples such as models of program logics, we would like to be able to formalise such reasoning.

A major approach to formalisation is via *dependent types*, used for example in the proof assistants Coq [18] and Agda [22]. Bizjak et al. [8], following earlier work [5, 20], introduced guarded dependent type theory (GDTT), integrating the $\triangleright$ type-former into a dependently typed calculus, and supporting the definition of guarded recursive types as fixed-points of functions on universes, and guarded recursive operations on these types.

We wish to formalise non-trivial theorems about equality between guarded recursive constructions, but such arguments often cannot be accommodated within *intensional* Martin-Löf type theory. For example, we may need to be able to reason about the extensions of streams in order to prove the equality of different stream functions. Hence GDTT includes an equality reflection rule, which is well known to make type checking undecidable. This problem is close to well-known problems with functional extensionality [13, Sec. 3.1.3], and indeed this analogy can be developed. Just as functional extensionality involves mapping terms of type $(x : A) \to \mathsf{Id}\, B\, (f\, x)\, (g\, x)$ to proofs of $\mathsf{Id}\, (A \to B)\, f\, g$, extensionality for guarded recursion requires an extensionality principle for later types, namely the ability to map terms of type $\triangleright\, \mathsf{Id}\, A\, t\, u$ to proofs of $\mathsf{Id}\, (\triangleright A)\, (\mathsf{next}\, t)\, (\mathsf{next}\, u)$, where next is the constructor for $\triangleright$. These types are isomorphic in the intended model, the presheaf category $\widehat{\omega}$ known as the *topos of trees*, and so in GDTT their equality was asserted as an axiom. But in a calculus without equality reflection we cannot merely assert such axioms without losing canonicity.

Cubical type theory (CTT) [10] is a new type theory with a computational interpretation of functional extensionality but without equality reflection, and hence is a candidate for extension with guarded recursion, so that we may formalise our arguments without incurring the disadvantages of fully extensional identity types. CTT was developed primarily to provide a computational interpretation of the univalence axiom of Homotopy Type Theory [26]. The most important novelty of CTT is the replacement of inductively defined identity types by *paths*, which can be seen as maps from an abstract interval $\mathbb{I}$, and are introduced and eliminated much like functions. CTT can be extended with identity types which model all rules of standard Martin-Löf type theory [10, Sec. 9.1], but these are equivalent to path types, and in our paper it suffices to work with path types only. CTT has sound denotational semantics in (fibrations in) *cubical sets*, a presheaf category that is used to model homotopy types. Many basic syntactic properties of CTT, such as the decidability of type checking, and canonicity for base types, are yet to be proved, but a type checker has been implemented[1] that confers some confidence in such properties.

In Sec. 2 of this paper we propose *guarded cubical type theory* (GCTT), a combination of the two type theories[2] which supports non-trivial proofs about guarded recursive types via path equality, while retaining the potential for good syntactic properties such as decidable type-checking and canonicity. In particular, just as a term can be defined in CTT to witness functional extensionality, a term can be defined in GCTT to witness extensionality for later types. Further, we use elements of the interval of CTT to annotate fixed-points, and hence control their unfoldings. This ensures that fixed-points are path equal, but not judgementally equal, to their unfoldings, and hence prevents infinite unfoldings, an obvious source of non-termination in any calculus with infinite constructions. The resulting calculus is shown via examples to be useful for reasoning about guarded recursive operations; we also view it as potentially significant from the point of view of CTT, extending its expressivity as a basis for formalisation.

In Sec. 3 we give sound semantics to this type theory via the presheaf category over the product of the categories used to define semantics for GDTT and CTT. This requires considerable work to ensure that the constructions of the two type theories remain sound in the new category, particularly the glueing and universe of CTT. The key technical challenge is to ensure that the $\triangleright$ type-former supports the *compositions* that all types must carry in the semantics of CTT.

[1] `https://github.com/mortberg/cubicaltt`

[2] with the exception of the *clock quantification* of GDTT, which we leave to future work.

We have implemented a prototype type-checker for this extended type theory[3], which provides confidence in the type theory's syntactic properties. All examples in this paper, and many others, have been formalised in this type checker.

For reasons of space many details and proofs are omitted from this paper, but are included in a technical appendix[4].

2 Guarded Cubical Type Theory

This section introduces guarded cubical type theory (GCTT), and presents examples of how it can be used to prove properties of guarded recursive constructions.

2.1 Cubical Type Theory

We first give a brief overview of *cubical type theory*[5] (CTT) [10]. We start with a standard dependent type theory with Π, Σ, natural numbers, and a Russell-style universe:

$$
\begin{array}{lcll}
\Gamma, \Delta & ::= & () \mid \Gamma, x : A & \text{Contexts} \\
t, u, A, B & ::= & x \mid \lambda x : A.t \mid t\, u \mid (x : A) \to B & \Pi\text{-types} \\
& \mid & (t, u) \mid t.1 \mid t.2 \mid (x : A) \times B & \Sigma\text{-types} \\
& \mid & 0 \mid \mathsf{s}\, t \mid \mathsf{natrec}\, t\, u \mid \mathsf{N} & \text{Natural numbers} \\
& \mid & \mathsf{U} & \text{Universe}
\end{array}
$$

We adhere to the usual conventions of considering terms and types up to α-equality, and writing $A \to B$, respectively $A \times B$, for non-dependent Π and Σ-types. We use the symbol '=' for judgemental equality.

The central novelty of CTT is its treatment of equality. Instead of the inductively defined identity types of intensional Martin-Löf type theory [17], CTT has *paths*. The paths between two terms t, u of type A form a sort of function space, intuitively that of continuous maps from some interval $\mathbb{I}$ to A, with endpoints t and u. Rather than defining the interval $\mathbb{I}$ concretely as the unit interval $[0, 1] \subseteq \mathbb{R}$, it is defined as the *free De Morgan algebra* on a discrete infinite set of names $\{i, j, k, \dots\}$. A De Morgan algebra is a bounded distributive lattice with an involution $1 - \cdot$ satisfying the De Morgan laws

$$1 - (i \wedge j) = (1 - i) \vee (1 - j), \qquad 1 - (i \vee j) = (1 - i) \wedge (1 - j).$$

The interval $[0, 1] \subseteq \mathbb{R}$, with min, max and $1 - \cdot$, is an example of a De Morgan algebra.

The syntax for elements of $\mathbb{I}$ is:

$$r, s \ ::= \ 0 \mid 1 \mid i \mid 1 - r \mid r \wedge s \mid r \vee s.$$

0 and 1 represent the endpoints of the interval. We extend the definition of contexts to allow introduction of a new name:

$$\Gamma, \Delta \ ::= \ \cdots \mid \Gamma, i : \mathbb{I}.$$

The judgement $\Gamma \vdash r : \mathbb{I}$ means that r draws its names from Γ. Despite this notation, $\mathbb{I}$ is not a first-class type. Path types and their elements are defined by the rules in Fig. 1. *Path abstraction*, $\langle i \rangle\, t$, and *path application*, $t\, r$, are analogous to λ-abstraction and function application, and support the familiar β-equality $(\langle i \rangle\, t)\, r = t[r/i]$ and η-equality $\langle i \rangle\, t\, i = t$. There are two

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Path}\ A\ t\ u}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, i : \mathbb{I} \vdash t : A}{\Gamma \vdash \langle i \rangle\, t : \mathsf{Path}\ A\ t[0/i]\ t[1/i]} \qquad\qquad \frac{\Gamma \vdash t : \mathsf{Path}\ A\ u\ s \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash t\, r : A}$$

Figure 1: Typing rules for path types.

additional judgemental equalities for paths, regarding their endpoints: given $p : \mathsf{Path}\ A\ t\ u$ we have $p\,0 = t$ and $p\,1 = u$.

Paths provide a notion of identity which is more extensional than that of intensional Martin-Löf identity types, as exemplified by the proof term for functional extensionality:

$$\mathsf{funext}\ f\ g \triangleq \lambda p.\, \langle i \rangle\, \lambda x.\, p\, x\, i\ :\ ((x : A) \to \mathsf{Path}\ B\ (f\, x)\ (g\, x)) \to \mathsf{Path}\ (A \to B)\ f\ g.$$

The rules above suffice to ensure that path equality is reflexive, symmetric, and a congruence, but we also need it to be transitive and, where the underlying type is the universe, to support a notion of transport. This is done via *(Kan) composition operations*.

To define these we need the *face lattice*, $\mathbb{F}$, defined as the free distributive lattice on the symbols $(i = 0)$ and $(i = 1)$ for all names i, quotiented by the relation $(i = 0) \wedge (i = 1) = 0_{\mathbb{F}}$. The syntax for elements of $\mathbb{F}$ is:

$$\varphi, \psi\ ::=\ 0_{\mathbb{F}} \mid 1_{\mathbb{F}} \mid (i = 0) \mid (i = 1) \mid \varphi \wedge \psi \mid \varphi \vee \psi.$$

As with the interval, $\mathbb{F}$ is not a first-class type, but the judgement $\Gamma \vdash \varphi : \mathbb{F}$ asserts that φ draws its names from Γ. We also have the judgement $\Gamma \vdash \varphi = \psi : \mathbb{F}$ which asserts the equality of φ and ψ in the face lattice. Contexts can be restricted by elements of $\mathbb{F}$:

$$\Gamma, \Delta\ ::=\ \cdots \mid \Gamma, \varphi.$$

Such a restriction affects equality judgements so that, for example, $\Gamma, \varphi \vdash \psi_1 = \psi_2 : \mathbb{F}$ is equivalent to $\Gamma \vdash \varphi \wedge \psi_1 = \varphi \wedge \psi_2 : \mathbb{F}$

We write $\Gamma \vdash t : A[\varphi \mapsto u]$ as an abbreviation for the two judgements $\Gamma \vdash t : A$ and $\Gamma, \varphi \vdash t = u : A$, noting the restriction with φ in the equality judgement. Now the composition operator is defined by the typing and equality rule

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A[0/i][\varphi \mapsto u[0/i]]}{\Gamma \vdash \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0 : A[1/i][\varphi \mapsto u[1/i]]}.$$

A simple use of composition is to implement the transport operation for Path types

$$\mathsf{transp}^i\ A\ a\ \triangleq\ \mathsf{comp}^i\ A\ [0_{\mathbb{F}} \mapsto []]\ a\ :\ A[1/i],$$

where a has type $A[0/i]$. The notation $[]$ stands for an empty *system*. In general a system is a list of pairs of faces and terms, and it defines an element of a type by giving the individual components at each face. We extend the syntax as follows:

$$t, u, A, B\ ::=\ \cdots \mid [\varphi_1\ t_1, \ldots, \varphi_n\ t_n].$$

[3] `http://github.com/hansbugge/cubicaltt/tree/gcubical`
[4] `http://cs.au.dk/~birke/papers/gdtt-cubical-technical-appendix.pdf`
[5] `http://www.cse.chalmers.se/~coquand/selfcontained.pdf` is a self-contained presentation of CTT.

$$\frac{\Gamma \vdash}{\vdash \cdot : \Gamma \nrightarrow \cdot} \qquad\qquad \frac{\vdash \xi : \Gamma \nrightarrow \Gamma' \qquad \Gamma \vdash t : \triangleright \xi.A}{\vdash \xi\,[x \leftarrow t] : \Gamma \nrightarrow \Gamma', x : A}$$

Figure 2: Formation rules for delayed substitutions.

Below we see two of the rules for systems; they ensure that the components of a system agree where the faces overlap, and that all the cases possible in the current context are covered:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash \varphi_1 \vee \ldots \vee \varphi_n = 1_{\mathbb{F}} : \mathbb{F} \qquad \Gamma, \varphi_i \vdash t_i : A \qquad \Gamma, \varphi_i \wedge \varphi_j \vdash t_i = t_j : A \qquad i,j = 1 \ldots n}{\Gamma \vdash [\varphi_1\ t_1, \ldots, \varphi_n\ t_n] : A}$$

$$\frac{\Gamma \vdash [\varphi_1\ t_1, \ldots, \varphi_n\ t_n] : A \qquad \Gamma \vdash \varphi_i = 1_{\mathbb{F}} : \mathbb{F}}{\Gamma \vdash [\varphi_1\ t_1, \ldots, \varphi_n\ t_n] = t_i : A}$$

We will shorten $[\varphi_1 \vee \ldots \vee \varphi_n \mapsto [\varphi_1\ t_1, \ldots, \varphi_n\ t_n]]$ to $[\varphi_1 \mapsto t_1, \ldots, \varphi_n \mapsto t_n]$.

A non-trivial example of the use of systems is the proof that Path is transitive; given $p : \mathsf{Path}\ A\ a\ b$ and $q : \mathsf{Path}\ A\ b\ c$ we can define

$$\mathsf{transitivity}\ p\,q \triangleq \langle i \rangle\ \mathsf{comp}^j\ A\ [(i = 0) \mapsto a, (i = 1) \mapsto q\,j]\ (p\,i) : \mathsf{Path}\ A\ a\ c.$$

This builds a path between the appropriate endpoints because we have the equalities $\mathsf{comp}^j\ A\ [1_{\mathbb{F}} \mapsto a]\ (p\,0) = a$ and $\mathsf{comp}^j\ A\ [1_{\mathbb{F}} \mapsto q\,j]\ (p\,1) = q\,1 = c$.

For reasons of space we have omitted the descriptions of some features of CTT, such as glueing, and the further judgemental equalities for terms of the form $\mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0$ that depend on the structure of A.

2.2 Later Types

In Fig. 3 we present the 'later' types of guarded dependent type theory (GDTT) [8], with judgemental equalities in Figs. 4 and 5. Note that we do not add any new equation for the interaction of compositions with $\triangleright$; such an equation would be necessary if we were to add the eliminator prev for $\triangleright$, but this extension (which involves clock quantifiers) is left to further work. We delay the presentation of the fixed-point operation until the next section.

The typing rules use the *delayed substitutions* of GDTT, as defined in Fig. 2. Delayed substitutions resemble Haskell-style do-notation, or a delayed form of let-binding. If we have a term $t : \triangleright A$, we cannot access its contents 'now', but if we are defining a type or term that itself has some part that is available 'later', then this part *should* be able to use the contents of t. Therefore delayed substitutions allow terms of type $\triangleright A$ to be unwrapped by $\triangleright$ and next. As observed by Bizjak et al. [8] these constructions generalise the *applicative functor* [19] structure of 'later' types, by the definitions $\mathsf{pure}\ t \triangleq \mathsf{next}\,t$, and $f \circledast t \triangleq \mathsf{next}\,[f' \leftarrow f, t' \leftarrow t]\,.\,f'\,t'$, as well as a generalisation of the $\circledast$ operation from simple functions to Π-types. We here make the new observation that delayed substitutions can express the function $\widehat{\triangleright} : \triangleright \mathsf{U} \to \mathsf{U}$, introduced by Birkedal and Møgelberg [4] to express guarded recursive types as fixed-points on universes, as $\lambda u.\triangleright[u' \leftarrow u].u'$; see for example the definition of streams in Sec. 2.4.

Example 1. In GDTT it is essential that we can convert terms of type $\triangleright\xi.\,\mathsf{Id}_A\ t\ u$ into terms of type $\mathsf{Id}_{\triangleright\xi.A}\ (\mathsf{next}\,\xi.\,t)\ (\mathsf{next}\,\xi.\,u)$, as it is essential for *Löb induction*, the technique of proof by

$$\frac{\Gamma, \Gamma' \vdash A \qquad \vdash \xi : \Gamma \twoheadrightarrow \Gamma'}{\Gamma \vdash \triangleright \xi . A} \qquad \frac{\Gamma, \Gamma' \vdash A : \mathsf{U} \qquad \vdash \xi : \Gamma \twoheadrightarrow \Gamma'}{\Gamma \vdash \triangleright \xi . A : \mathsf{U}} \qquad \frac{\Gamma, \Gamma' \vdash t : A \qquad \vdash \xi : \Gamma \twoheadrightarrow \Gamma'}{\Gamma \vdash \mathsf{next}\, \xi . t : \triangleright \xi . A}$$

Figure 3: Typing rules for later types.

$$\frac{\vdash \xi\, [x \leftarrow t] : \Gamma \twoheadrightarrow \Gamma', x : B \qquad \Gamma, \Gamma' \vdash A}{\Gamma \vdash \triangleright \xi\, [x \leftarrow t]\, . A = \triangleright \xi . A}$$

$$\frac{\vdash \xi\, [x \leftarrow t, y \leftarrow u]\, \xi' : \Gamma \twoheadrightarrow \Gamma', x : B, y : C, \Gamma'' \qquad \Gamma, \Gamma' \vdash C \qquad \Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash A}{\Gamma \vdash \triangleright \xi\, [x \leftarrow t, y \leftarrow u]\, \xi' . A = \triangleright \xi\, [y \leftarrow u, x \leftarrow t]\, \xi' . A}$$

$$\frac{\vdash \xi : \Gamma \twoheadrightarrow \Gamma' \qquad \Gamma, \Gamma', x : B \vdash A \qquad \Gamma, \Gamma' \vdash t : B}{\Gamma \vdash \triangleright \xi\, [x \leftarrow \mathsf{next}\, \xi . t]\, . A = \triangleright \xi . A[t/x]}$$

Figure 4: Type equality rules for later types (congruence and equivalence rules are omitted).

$$\frac{\vdash \xi\, [x \leftarrow t] : \Gamma \twoheadrightarrow \Gamma', x : B \qquad \Gamma, \Gamma' \vdash u : A}{\Gamma \vdash \mathsf{next}\, \xi\, [x \leftarrow t]\, . u = \mathsf{next}\, \xi . u : \triangleright \xi . A}$$

$$\frac{\vdash \xi\, [x \leftarrow t, y \leftarrow u]\, \xi' : \Gamma \twoheadrightarrow \Gamma', x : B, y : C, \Gamma'' \qquad \Gamma, \Gamma' \vdash C \qquad \Gamma, \Gamma', x : B, y : C, \Gamma'' \vdash v : A}{\Gamma \vdash \mathsf{next}\, \xi\, [x \leftarrow t, y \leftarrow u]\, \xi' . v = \mathsf{next}\, \xi\, [y \leftarrow u, x \leftarrow t]\, \xi' . v : \triangleright \xi\, [x \leftarrow t, y \leftarrow u]\, \xi' . A}$$

$$\frac{\vdash \xi : \Gamma \twoheadrightarrow \Gamma' \qquad \Gamma, \Gamma', x : B \vdash u : A \qquad \Gamma, \Gamma' \vdash t : B}{\Gamma \vdash \mathsf{next}\, \xi\, [x \leftarrow \mathsf{next}\, \xi . t]\, . u = \mathsf{next}\, \xi . u[t/x] : \triangleright \xi . A[t/x]} \qquad \frac{\Gamma \vdash t : \triangleright \xi . A}{\Gamma \vdash \mathsf{next}\, \xi\, [x \leftarrow t]\, . x = t : \triangleright \xi . A}$$

Figure 5: Term equality rules for later types. We omit congruence and equivalence rules, and the rules for terms of type U, which reflect the type equality rules of Fig. 4.

$$\frac{\Gamma \vdash r : \mathbb{I} \qquad \Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{dfix}^r\, x.t : \triangleright A} \qquad \frac{\Gamma, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{dfix}^1\, x.t = \mathsf{next}\, t[\mathsf{dfix}^0\, x.t/x] : \triangleright A}.$$

Figure 6: Typing and equality rules for the delayed fixed-point

guarded recursion where we assume $\triangleright p$, deduce p, and hence may conclude p with no assumptions. This is achieved in GDTT by postulating as an axiom the following judgemental equality:

$$\mathsf{Id}_{\triangleright \xi. A}\, (\mathsf{next}\, \xi.\, t)\ (\mathsf{next}\, \xi.\, u)\ =\ \triangleright \xi.\, \mathsf{Id}_A\ t\ u \tag{1}$$

A term from left-to-right of (1) can be defined using the J-eliminator for identity types, but the more useful direction is right-to-left, as proofs of equality by Löb induction involve assuming that we later have a path, then converting this into a path on later types. In fact in GCTT we can define a term with the desired type:

$$\lambda p. \langle i \rangle\, \mathsf{next}\, \xi [p' \leftarrow p].\, p'\, i\ :\ (\triangleright \xi.\, \mathsf{Path}\, A\, t\, u) \to \mathsf{Path}\, (\triangleright \xi. A)\, (\mathsf{next}\, \xi.\, t)\, (\mathsf{next}\, \xi.\, u). \tag{2}$$

Note the similarity of this term and type with that of funext, for functional extensionality, presented on page 4. Indeed we claim that (2) provides a computational interpretation of extensionality for later types.

2.3 Fixed Points

In this section we complete the presentation of GCTT by addressing fixed points. In GDTT there are fixed-point constructions $\mathsf{fix}\, x.t$ with the judgemental equality $\mathsf{fix}\, x.t = t[\mathsf{next}\, \mathsf{fix}\, x.t/x]$. In GCTT we want decidable type checking, including decidable judgemental equality, and so we cannot admit such an unrestricted unfolding rule. Our solution it that fixed points should not be judgementally equal to their unfoldings, but merely *path equal*. We achieve this by decorating the fixed-point combinator with an interval element which specifies the position on this path. The 0-endpoint of the path is the stuck fixed-point term, while the 1-endpoint is the same term unfolded once. However this threatens canonicity for base types: if we allow stuck fixed-points in our calculus, we could have stuck closed terms $\mathsf{fix}^i\, x.t$ inhabiting N. To avoid this, we introduce the *delayed* fixed-point combinator dfix, which produces a term 'later' instead of a term 'now'. Its typing rule, and notion of equality, is given in Fig. 6. We will write $\mathsf{fix}^r\, x.t$ for $t[\mathsf{dfix}^r\, x.t/x]$, $\mathsf{fix}\, x.t$ for $\mathsf{fix}^0\, x.t$, and $\mathsf{dfix}\, x.t$ for $\mathsf{dfix}^0\, x.t$.

Lemma 2 (Canonical unfold lemma)**.** *For any term $\Gamma, x : \triangleright A \vdash t : A$ there is a path between $\mathsf{fix}\, x.t$ and $t[\mathsf{next}\, \mathsf{fix}\, x.t/x]$, given by the term $\langle i \rangle\, \mathsf{fix}^i\, x.t$.*

Transitivity of paths (via compositions) ensures that $\mathsf{fix}\, x.t$ is path equal to any number of fixed-point unfoldings of itself.

A term a of type A is said to be a *guarded fixed point* of a function $f : \triangleright A \to A$ if there is a path from a to $f(\mathsf{next}\, a)$.

Proposition 3 (Unique guarded fixed points)**.** *Any guarded fixed-point a of a term $f : \triangleright A \to A$ is path equal to $\mathsf{fix}\, x.f\, x$.*

Proof. Given $p : \mathsf{Path}\, A\ a\ (f\, (\mathsf{next}\, a))$, we proceed by Löb induction, i.e., by assuming $\mathsf{ih} : \triangleright(\mathsf{Path}\, A\ a\ (\mathsf{fix}\, x.f\, x))$. We can define a path

$$s \triangleq \langle i \rangle\, f(\mathsf{next}\, [q \leftarrow \mathsf{ih}]\, .\, q\, i)\ :\ \mathsf{Path}\, A\ (f(\mathsf{next}\, a))\ (f(\mathsf{next}\, \mathsf{fix}\, x.f\, x)),$$

which is well-typed because the type of the variable q ensures that $q\,0$ is judgementally equal to a, resp. $q\,1$ and $\mathsf{fix}\,x.f\,x$. Note that we here implicitly use the extensionality principle for later (2). We compose s with p, and then with the inverse of the canonical unfold lemma of Lem. 2, to obtain our path from a to $\mathsf{fix}\,x.f\,x$. We can write out our full proof term, where p^{-1} is the inverse path of p, as

$$\mathsf{fix}\,\mathsf{ih}\,.\,\langle i\rangle\,\mathsf{comp}^j\ A\ [(i=0)\mapsto p^{-1}, (i=1)\mapsto f(\mathsf{dfix}^{1-j}\,x.f\,x)]\ (f(\mathsf{next}\,[q\leftarrow \mathsf{ih}]\,.\,q\,i)). \qquad \square$$

2.4 Programming and Proving with Guarded Recursive Types

In this section we show some simple examples of programming with guarded recursion, and prove properties of our programs using Löb induction.

Streams. The type of guarded recursive streams in GCTT, as with GDTT, are defined as fixed points on the universe:

$$\mathsf{Str}_A \triangleq \mathsf{fix}\,x.A \times \triangleright[y\leftarrow x].y$$

Note the use of a delayed substitution to transform a term of type $\triangleright\,\mathsf{U}$ to one of type U, as discussed at the start of Sec. 2.2. Desugaring to restate this in terms of dfix, we have

$$\mathsf{Str}_A \;=\; A \times \triangleright[y\leftarrow \mathsf{dfix}^0\,x.A\times\triangleright[y\leftarrow x].y].y$$

The head function $\mathsf{hd} : \mathsf{Str}_A \to A$ is the first projection. The tail function, however, cannot be the second projection, since this yields a term of type

$$\triangleright\left[y\leftarrow \mathsf{dfix}^0\,x.A\times\triangleright\left[y\leftarrow x\right].y\right].y \tag{3}$$

rather than the desired $\triangleright\,\mathsf{Str}_A$. However we are not far off; $\triangleright\,\mathsf{Str}_A$ is judgementally equal to $\triangleright\left[y\leftarrow \mathsf{dfix}^1\,x.A\times\triangleright\left[y\leftarrow x\right].y\right].y$, which is the same term as (3), apart from endpoint 1 replacing 0. The canonical unfold lemma (Lem. 2) tells us that we can build a path in U from Str_A to $A\times\triangleright\,\mathsf{Str}_A$; call this path $\langle i\rangle\,\mathsf{Str}_A^i$. Then we can transport between these types:

$$\mathsf{unfold}\ s \triangleq \mathsf{transp}^i\ \mathsf{Str}_A^i\ s \qquad\qquad \mathsf{fold}\ s \triangleq \mathsf{transp}^i\ \mathsf{Str}_A^{1-i}\ s$$

Note that the compositions of these two operations are path equal to identity functions, but not judgementally equal. We can now obtain the desired tail function $\mathsf{tl} : \mathsf{Str}_A \to \triangleright\,\mathsf{Str}_A$ by composing the second projection with unfold, so $\mathsf{tl}\ s \triangleq (\mathsf{unfold}\ s).2$. Similarly we can define the stream constructor cons (written infix as ::) by using fold:

$$\mathsf{cons} \triangleq \lambda a, s.\,\mathsf{fold}\,(a,s)\ :\ A\to\triangleright\,\mathsf{Str}_A\to\mathsf{Str}_A\,.$$

We now turn to higher order functions on streams. We define $\mathsf{zipWith} : (A\to B\to C)\to \mathsf{Str}_A\to\mathsf{Str}_B\to\mathsf{Str}_C$, the stream function which maps a binary function on two input streams to produce an output stream, as

$$\mathsf{zipWith}\ f \triangleq \mathsf{fix}\,z.\lambda s_1,s_2.f\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\ ::\ \mathsf{next}\left[\begin{array}{l} z'\leftarrow z \\ t_1\leftarrow \mathsf{tl}\ s_1 \\ t_2\leftarrow \mathsf{tl}\ s_2\end{array}\right].\,z'\,t_1\,t_2.$$

Of course $\mathsf{zipWith}$ is definable even with simple types and $\triangleright$, but in GCTT we can go further and prove properties about the function:

Proposition 4 (zipWith preserves commutativity)**.** *If $f : A \to A \to B$ is commutative, then* $\mathsf{zipWith}\ f : \mathsf{Str}_A \to \mathsf{Str}_A \to \mathsf{Str}_B$ *is commutative.*

Proof. Let $\mathsf{c} : (a_1 : A) \to (a_2 : A) \to \mathsf{Path}\, B\ (f\, a_1\, a_2)\ (f\, a_2\, a_1)$ witness commutativity of f. We proceed by Löb induction, i.e., by assuming

$$\mathsf{ih} : \triangleright((s_1 : \mathsf{Str}_A) \to (s_2 : \mathsf{Str}_A) \to \mathsf{Path}\, B\ (\mathsf{zipWith}\ f\, s_1\, s_2)\ (\mathsf{zipWith}\ f\, s_2\, s_1))\,.$$

Let $i : \mathbb{I}$ be a fresh name, and $s_1, s_2 : \mathsf{Str}_A$. Our aim is to construct a stream v which is $\mathsf{zipWith}\ f\, s_1\, s_2$ when substituting 0 for i, and $\mathsf{zipWith}\ f\, s_2\, s_1$ when substituting 1 for i. An initial attempt at this proof is the term

$$v \triangleq \mathsf{c}\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\,i\ ::\ \mathsf{next}\left[\begin{array}{l} q \leftarrow \mathsf{ih} \\ t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\, q\, t_1\, t_2\, i\ :\ \mathsf{Str}_B,$$

which is equal to

$$f\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\ ::\ \mathsf{next}\left[\begin{array}{l} t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\ \mathsf{zipWith}\ f\, t_1\, t_2$$

when substituting 0 for i, which is $\mathsf{zipWith}\ f\, s_1\, s_2$, but *unfolded once.* Similarly, $v[1/i]$ is $\mathsf{zipWith}\ f\, s_2\, s_1$ unfolded once. Let $\langle j \rangle\, \mathsf{zipWith}^j$ be the canonical unfold lemma associated with zipWith (see Lem. 2). We can now finish the proof by composing v with (the inverse of) the canonical unfold lemma. Diagrammatically, with i along the horizontal axis and j along the vertical:

$$\begin{array}{ccc} \mathsf{zipWith}\ f\, s_1\, s_2 & \dashrightarrow & \mathsf{zipWith}\ f\, s_2\, s_1 \\ \Big\uparrow {\scriptstyle \mathsf{zipWith}^{1-j}\ f\, s_1\, s_2} & & \Big\uparrow {\scriptstyle \mathsf{zipWith}^{1-j}\ f\, s_2\, s_1} \\ f\,(\mathsf{hd}\ s_1)\,(\mathsf{hd}\ s_2)\ :: \\ \mathsf{next}\left[\begin{array}{l} t_1 \leftarrow \mathsf{tl}\ s_1 \\ t_2 \leftarrow \mathsf{tl}\ s_2 \end{array}\right].\ \mathsf{zipWith}\ f\, t_1\, t_2 & \xrightarrow{\ v\ } & f\,(\mathsf{hd}\ s_2)\,(\mathsf{hd}\ s_1)\ :: \\ & & \mathsf{next}\left[\begin{array}{l} t_2 \leftarrow \mathsf{tl}\ s_2 \\ t_1 \leftarrow \mathsf{tl}\ s_1 \end{array}\right].\ \mathsf{zipWith}\ f\, t_2\, t_1 \end{array}$$

The complete proof term, in the language of the type checker, can be found in Appendix A. □

Guarded recursive types with negative variance. A key feature of guarded recursive types are that they support *negative* occurrences of recursion variables. This is important for applications to models of program logics [6]. Here we consider a simple example of a negative variance recursive type, namely $\mathsf{Rec}_A \triangleq \mathsf{fix}\, x.(\triangleright[x' \leftarrow x].x') \to A$, which is path equal to $\triangleright\,\mathsf{Rec}_A \to A$. As a simple demonstration of the expressiveness we gain from negative guarded recursive types, we define a guarded variant of Curry's Y combinator:

$$\begin{array}{lcll} \Delta & \triangleq & \lambda x.f(\mathsf{next}[x' \leftarrow x].\,((\mathsf{unfold}\, x')x)) & :\ \ \triangleright\,\mathsf{Rec}_A \to A \\ \mathsf{Y} & \triangleq & \lambda f.\Delta(\mathsf{next}\,\mathsf{fold}\,\Delta) & :\ \ (\triangleright A \to A) \to A, \end{array}$$

where fold and unfold are the transports along the path between Rec_A and $\triangleright\,\mathsf{Rec}_A \to A$. As with zipWith, Y can be defined with simple types and $\triangleright$ [1]; what is new to GCTT is that we can also prove properties about it:

Proposition 5 (Y is a guarded fixed-point combinator)**.** $\mathsf{Y}\, f$ *is path equal to* $f\,(\mathsf{next}(\mathsf{Y}\, f))$*, for any* $f : \triangleright A \to A$*. Therefore, by Prop. 3,* Y *is path equal to* fix.

Proof. $\mathsf{Y}\, f$ simplifies to $f\,(\mathsf{next}(\mathsf{unfold}\,(\mathsf{fold}\,\Delta)\,(\mathsf{next}\,\mathsf{fold}\,\Delta)))$, and unfold $(\mathsf{fold}\,\Delta)$ is path equal to Δ. A congruence over this path yields our path between $\mathsf{Y}\, f$ and $f(\mathsf{next}(\mathsf{Y}\, f))$. □

3 Semantics

In this section we sketch the semantics of GCTT. The semantics is based on the category $\widehat{\mathcal{C} \times \omega}$ of presheaves on the category $\mathcal{C} \times \omega$, where $\mathcal{C}$ is the *category of cubes* [10] and ω is the poset of natural numbers. The category of cubes is the opposite of the Kleisli category of the free De Morgan algebra monad on finite sets. More concretely, given a countably infinite set of names $i, j, k, \ldots$, $\mathcal{C}$ has as objects finite sets of names I, J. A morphism $I \to J \in \mathcal{C}$ is a *function* $J \to \mathbf{DM}\,(I)$, where $\mathbf{DM}\,(I)$ is the free De Morgan algebra with generators I.

Following the approach of Cohen et al. [10], contexts of GCTT will be interpreted as objects of $\widehat{\mathcal{C} \times \omega}$. Types in context Γ will be interpreted as pairs (A, c_A) of a presheaf A on the category of elements of Γ and a *composition structure* c_A. We call such a pair a *fibrant* type.

To aid in defining what a composition structure is, and in showing that composition structure is preserved by all the necessary type constructions, we will make use of the internal language of $\widehat{\mathcal{C} \times \omega}$ in the form of *dependent predicate logic*; see for example Phoa [24, App. I].

A type of GCTT in context Γ will then be interpreted as a pair of a type $\Gamma \vdash A$ in the internal language of $\widehat{\mathcal{C} \times \omega}$, and a composition structure c_A, where c_A is a term in the internal language of a specific type $\Phi(\Gamma; A)$, which we define below after introducing the necessary constructs. Terms of GCTT will be interpreted as terms of the internal language. We use *categories with families* [12] as our notion of a model. Due to space limits we omit the precise definition of the category with families here, and refer to the online technical appendix.

The semantics is split into several parts, which provide semantics at different levels of generality.

1. We first show that every presheaf topos with a non-trivial internal De Morgan algebra $\mathbb{I}$ satisfying the disjunction property can be used to give semantics to the subset of the cubical type theory CTT without glueing and the universe. We further show that, for any category $\mathbb{D}$, the category of presheaves on $\mathcal{C} \times \mathbb{D}$ has an interval $\mathbb{I}$, which is the inclusion of the interval in presheaves over the category of cubes $\mathcal{C}$.

2. We then extend the semantics to include glueing and universes. We show that the topos of presheaves $\mathcal{C} \times \mathbb{D}$ for any category $\mathbb{D}$ with an initial object can be used to give semantics to the entire cubical type theory.

3. Finally, we show that the category of presheaves on $\mathcal{C} \times \omega$ gives semantics to delayed substitutions and fixed points. Using these and some additional properties of the delayed substitutions we show in the internal language of $\widehat{\mathcal{C} \times \omega}$ that $\triangleright \xi.A$ has composition whenever A has composition.

Combining all three, we give semantics to GCTT in $\widehat{\mathcal{C} \times \omega}$.

3.1 Model of CTT Without Glueing and the Universe

Let $\mathcal{E}$ be a topos with a natural numbers object, and let $\mathbb{I}$ be a De Morgan algebra internal to $\mathcal{E}$ which satisfies the *finitary disjunction property*, i.e.,

$$(i \vee j) = 1 \implies (i = 1) \vee (j = 1), \quad \text{and} \quad \neg(0 = 1).$$

Faces. Using the interval $\mathbb{I}$ we define the type $\mathbb{F}$ as the image of the function $\cdot = 1 : \mathbb{I} \to \Omega$, where Ω is the subobject classifier. More precisely, $\mathbb{F}$ is the subset type

$$\mathbb{F} \triangleq \{p : \Omega \mid \exists (i : \mathbb{I}), p = (i = 1)\}$$

We will implicitly use the inclusion $\mathbb{F} \to \Omega$. The following lemma states in particular that the inclusion is compatible with all the lattice operations, so omitting it is justified. The disjunction property is crucial for validity of this lemma.

Lemma 6.

- $\mathbb{F}$ *is a lattice for operations inherited from* Ω.
- *The corestriction* $\cdot = 1 : \mathbb{I} \to \mathbb{F}$ *is a lattice homomorphism. It is not injective in general.*

Given $\Gamma \vdash \varphi : \mathbb{F}$, we write $[\varphi] \triangleq \mathrm{Id}_{\mathbb{F}}(\varphi, \top)$. Given $\Gamma \vdash A$ and $\Gamma \vdash \varphi : \mathbb{F}$ a *partial element* of type A of *extent* φ is a term t of type $\Gamma \vdash t : \Pi(p : [\varphi]).A$. If we are in a context with $p : [\varphi]$, then we will treat such a partial element t as a term of type A, leaving implicit the application to the proof p, i.e., we will treat t as $t\,p$. We will often write $\Gamma, [\varphi]$ instead of $\Gamma, p : [\varphi]$ when we do not mention the proof term p explicitly in the rest of the judgement. This is justified since inhabitants of $[\varphi]$ are unique up to judgemental equality (recall that dependent predicate logic is a logic over an extensional dependent type theory). Given $\Gamma, p : [\varphi] \vdash B$ we write B^{φ} for the dependent function space $\Pi(p : [\varphi]).B$ and again leave the proof p implicit.

For a term $\Gamma, p : [\varphi] \vdash u : A$ we define $A[\varphi \mapsto u] \triangleq \Sigma(a : A).\,(\mathrm{Id}_A(a, u))^{\varphi}$.

Compositions. Faces allow us to define the type of *compositions* $\Phi(\Gamma; A)$. Homotopically, compositions allow us to put a lid on a box [10]. Given $\Gamma \vdash A$ we define the corresponding type of compositions as

$$\Phi(\Gamma; A) \triangleq \Pi(\gamma : \mathbb{I} \to \Gamma)(\varphi : \mathbb{F})\,(u : \Pi(i : \mathbb{I}).\,(A(\gamma(i)))^{\varphi})\,. \\ A(\gamma(0))[\varphi \mapsto u(0)] \to A(\gamma(1))[\varphi \mapsto u(1)].$$

Here we treat the context Γ as a closed type. This is justified because there is a canonical bijection between contexts and closed types of the internal language. The notation $A(\gamma(i))$ means substitution along the (uncurried) γ.

Due to lack of space we do not show how the standard constructs of the type theory are interpreted. We only sketch how the following composition term is interpreted in terms of the composition in the model.

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A[0/i][\varphi \mapsto u[0/i]]}{\Gamma \vdash \mathsf{comp}^i\ A\ [\varphi \mapsto u]\ a_0 : A[1/i][\varphi \mapsto u[1/i]]}.$$

By assumption we have c_A of type $\Phi(\Gamma, i : \mathbb{I}; A)$ and u and a_0 are interpreted as terms in the internal language of the corresponding types. The interpretation of composition is the term

$$\gamma : \Gamma \vdash c_A\,(\lambda(i : \mathbb{I}).(\gamma, i))\,\varphi\,(\lambda(i : \mathbb{I})(p : [\varphi]).u)\,a_0 : A(\gamma(1))[\varphi \mapsto u(1)]$$

where we have omitted writing the proof $u(0) = a_0$ on $[\varphi]$.

Concrete models. The category of cubical sets has an internal interval type satisfying the disjunction property [10]. It is the functor mapping $I \in \mathcal{C}$ to $\mathbf{DM}\,(I)$. Since the theory of a De Morgan algebra with $0 \neq 1$ and the disjunction property is geometric [16, Section X.3] we have that for any topos $\mathcal{F}$ and geometric morphism $\varphi : \mathcal{F} \to \widehat{\mathcal{C}}$, $\varphi^*(\mathbb{I}) \in \mathcal{F}$ is a De Morgan algebra with the disjunction property[6]. In particular, given any category $\mathbb{D}$ there is a projection functor $\pi : \mathcal{C} \times \mathbb{D} \to \mathcal{C}$ which induces the (essential) geometric morphism $\pi^* \dashv \pi_* : \widehat{\mathcal{C} \times \mathbb{D}} \to \widehat{\mathcal{C}}$, where π^* is precomposition with π, and π_* takes limits along $\mathbb{D}$.

[6] A statement very close to this can be used as a characterisation of $\widehat{\mathcal{C}}$: this topos classifies the geometric theory of flat De Morgan algebras [25].

Summary. With the semantic structures developed thus far we can give semantics to the subset of CTT without glueing and the universe.

3.2 Adding Glueing and the Universe

The glueing construction [10, Sec. 6] is used to prove both fibrancy and, subsequently, univalence of the universe of fibrant types. Concretely, given

$$\Gamma \vdash \varphi : \mathbb{I} \qquad \Gamma, [\varphi] \vdash T \qquad \Gamma \vdash A \qquad \Gamma \vdash w : (T \to A)^{\varphi}$$

we define the type Glue $[\varphi \mapsto (T, w)]\ A$ in two steps. First we define the type[7]

$$Glue'_{\Gamma}(\varphi, T, A, w) \triangleq \sum_{a:A} \sum_{t:T^{\varphi}} \prod_{p:[\varphi]} wp(tp) = a.$$

For this type we have the following property $\Gamma, [\varphi] \vdash T \cong Glue'_{\Gamma}(\varphi, T, A, w)$. However, we need an equality, not an isomorphism, to obtain the correct typing rules. The technical appendix provides a general strictification lemma which allows us to define the type *Glue*.

To show that the type Glue $[\varphi \mapsto (T, w)]\ A$ is fibrant we need to additionally assume that the map $\varphi \mapsto \lambda\_.\varphi : \mathbb{F} \to (\mathbb{I} \to \mathbb{F})$ has an internal right adjoint $\forall$. Such a right adjoint exists in all toposes $\widehat{\mathcal{C} \times \mathbb{D}}$, for any small category $\mathbb{D}$ with an initial object.

Universe of fibrant types. Given a (Grothendieck) universe $\mathfrak{U}$ in the meta-theory, the Hofmann-Streicher universe [14] $\mathcal{U}^{\omega}$ in $\widehat{\mathcal{C} \times \omega}$ maps (I, n) to the set of functors valued in $\mathfrak{U}$ on the category of elements of $y(I, n)$, where y is the Yoneda embedding. As in Cohen et al. [10] we define the universe of *fibrant* types $\mathcal{U}_f^{\omega}$ by setting $\mathcal{U}_f^{\omega}(I, n)$ to be the set of fibrant types in context $y(I, n)$. The universe $\mathcal{U}_f^{\omega}$ satisfies the rules

$$\frac{\Gamma \vdash a : \mathcal{U} \qquad \vdash \mathbf{c} : \Phi(\Gamma; \mathrm{El}(a))}{\Gamma \vdash (\!|a, \mathbf{c}|\!) : \mathcal{U}_f} \qquad \frac{\Gamma \vdash a : \mathcal{U}_f}{\Gamma \vdash \mathrm{El}(a)} \qquad \frac{\Gamma \vdash a : \mathcal{U}_f}{\vdash \mathrm{Comp}(a) : \Phi(\Gamma; \mathrm{El}(a))}$$

Using the glueing operation, one shows that the universe of fibrant types is *itself* fibrant and, moreover, that it is univalent.

3.3 Adding the Later Type-Former

We now fix the site to be $\mathcal{C} \times \omega$. From the previous sections we know that $\widehat{\mathcal{C} \times \omega}$ gives semantics to CTT. The new constructs of GDTT are the $\triangleright$ type-former and its delayed substitutions, and guarded fixed points. Continuing to work in the internal language, we first show that the internal language of $\widehat{\mathcal{C} \times \omega}$ can be extended with these constructions, allowing interpretation of the subset of the type theory GDTT without clock quantification [8]. Due to lack of space we omit the details of this part, but do remark that $\triangleright$ is defined as

$$(\triangleright(X))(I, n) \begin{cases} \{\star\} & \text{if } n = 0 \\ X(I, m) & \text{if } n = m + 1 \end{cases}$$

The essence of this definition is that $\triangleright$ depends only on the "ω component" and ignores the "$\mathcal{C}$ component". Verification that all the rules of GDTT are satisfied is therefore very similar to the verification that the topos $\widehat{\omega}$ is a model of the same subset of GDTT.

[7] This type is already present in Kapulkin at al. [15, Thm 3.4.1].

The only additional property we need now is that $\triangleright$ preserves compositions, in the sense that if we have a delayed substitution $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$ and a type $\Gamma, \Gamma' \vdash A$ together with a closed term $\mathbf{c}_A$ of type $\Phi(\Gamma, \Gamma'; A)$ then we can construct $\mathbf{c}'_{\triangleright\xi.A}$ of type $\Phi(\Gamma; \triangleright\xi.A)$.

The following lemma uses the notion of a type $\Gamma \vdash A$ being *constant with respect to* ω. This notion is a natural generalisation to types-in-context of the property that a presheaf is in the image of the functor π^*. We refer to the online technical appendix for the precise definition. Here we only remark that the interval type $\mathbb{I}$ is constant with respect to ω, as is the type $\Gamma \vdash [\varphi]$ for any term $\Gamma \vdash \varphi : \mathbb{F}$.

Lemma 7. *Assume $\Gamma \vdash A$, $\Gamma, \Gamma', x : A \vdash B$ and $\vdash \xi : \Gamma \twoheadrightarrow \Gamma'$, and further that A is constant with respect to ω. Then the following two types are isomorphic*

$$\Gamma \vdash \triangleright\xi.\Pi(x : A).B \cong \Pi(x : A).\triangleright\xi.B \tag{4}$$

and the canonical morphism $\lambda f.\lambda x.\, \mathsf{next}\,[\xi, f' \leftarrow f]\,.\, f'\, x$ from left to right is an isomorphism.

Corollary 8. *If $\Gamma \vdash \varphi : \mathbb{F}$ then we have an isomorphism of types*

$$\Gamma \vdash \triangleright\xi.\Pi(p : [\varphi]).B \cong \Pi(x : [\varphi]).\triangleright\xi.B. \tag{5}$$

Lemma 9 ($\triangleright\xi$-types preserve compositions)**.** *If $\triangleright\xi.A$ is a well-formed type in context Γ and we have a composition term $\mathbf{c}_A : \Phi(\Gamma, \Gamma'; A)$, then there is a composition term $\mathbf{c} : \Phi(\Gamma; \triangleright\xi.A)$.*

Proof. We show the special case with an empty delayed substitution. For the more general proof we refer to the technical appendix. Assume we have a composition $\mathbf{c}_A : \Phi(\Gamma; A)$. Our goal is to find a term $\mathbf{c} : \Phi(\Gamma; \triangleright A)$, so we first introduce some variables:

$$\gamma : \mathbb{I} \to \Gamma \qquad \varphi : \mathbb{F} \qquad u : \Pi(i : \mathbb{I}).\, ((\triangleright A)(\gamma\, i))^{\varphi} \qquad a_0 : (\triangleright A)(\gamma\, 0)[\varphi \mapsto u\, 0].$$

Using the isomorphisms from Cor. 8 and Lem. 7 we obtain a term $\tilde{u} : \triangleright(\Pi(i : \mathbb{I}).(A(\gamma\, i))^{\varphi})$ isomorphic to u. We can now – almost – write the term

$$\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\, \mathbf{c}_A\, \gamma\, \varphi\, u'\, a'_0 \;:\; \triangleright(A(\gamma\, 1)), \tag{$*$}$$

what is missing is to check that $a'_0 = u'\, 0$ on the extent φ, so that we can legally apply $\mathbf{c}_A$; this is equivalent to saying that the type $\triangleright\,[u' \leftarrow \tilde{u}, a'_0 \leftarrow a_0]\,.\,\mathrm{Id}_{A(\gamma\, 0)}(a'_0, u'\, 0)^{\varphi}$ is inhabited. We transform this type as follows:

$$\begin{aligned} \triangleright\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\mathrm{Id}(a'_0, u'\, 0)^{\varphi} &\cong \left(\triangleright\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\mathrm{Id}(a'_0, u'\, 0)\right)^{\varphi} && \text{(Cor. 8)} \\ &= \left(\mathrm{Id}(\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].a'_0, \mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].u'\, 0)\right)^{\varphi} \\ &= (\mathrm{Id}(a_0, u\, 0))^{\varphi}, \end{aligned}$$

where the last equality uses that $\tilde{u}$ is defined using the inverse of $\lambda f\lambda x.\, \mathsf{next}\,[f' \leftarrow f]\,.\, f'\, x$ (Lem. 7). By assumption it is the case that $(\mathrm{Id}(a_0, u\, 0))^{\varphi}$ is inhabited, and therefore $(*)$ is well-defined. It remains only to check that $(*)$ is equal to $u\, 1$ on the extent φ, but this follows from the equalities of $\mathbf{c}_A$ and by the definition of $\tilde{u}$ (Lem. 7). Assuming φ, we have

$$\mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\, \mathbf{c}_A\, \gamma\, \varphi\, u'\, a'_0 = \mathsf{next}\left[\begin{array}{l} u' \leftarrow \tilde{u} \\ a'_0 \leftarrow a_0 \end{array}\right].\, u'\, 1 = u\, 1. \qquad \square$$

Summary. In this section we have highlighted the key ingredients that go into a sound interpretation of GCTT in $\widehat{\mathcal{C} \times \omega}$. For the precise statement of the interpretation of all the constructs, and the soundness theorem, we refer to the online technical appendix.

4 Conclusion

In this paper we have made the following contributions:

- We introduce guarded cubical type theory (GCTT), which combines features of cubical type theory (CTT) and guarded dependent type theory (GDTT). The path equality of CTT is shown to support reasoning about extensional properties of guarded recursive operations, and we use the interval of CTT to constrain the unfolding of fixed-points.
- We show that CTT can be modelled in any presheaf topos with an internal non-trivial De Morgan algebra with the disjunction property, an operator $\forall$, and a universe of fibrant types. Most of these constructions are done via the internal logic. We then show that a class of presheaf models of the form $\widehat{\mathcal{C} \times \mathbb{D}}$, for any category $\mathbb{D}$ with an initial object, satisfy the above axioms and hence gives rise to a model of CTT.
- We give semantics to GCTT in the topos of presheaves over $\mathcal{C} \times \omega$.

Further work. We wish to establish key *syntactic properties* of GCTT, namely decidable type-checking and canonicity for base types. Our prototype implementation establishes some confidence in these properties.

We wish to further extend GCTT with *clock quantification* [3], such as is present in GDTT. Clock quantification allows for the controlled elimination of the later type-former, and hence the encoding of first-class coinductive types via guarded recursive types. The generality of our approach to semantics in this paper should allow us to build a model by combining cubical sets with the presheaf model of GDTT with multiple clocks [7]. The main challenges lie in ensuring decidable type checking (GDTT relies on certain rules involving clock quantifiers which seem difficult to implement), and solving the *coherence problem* for clock substitution.

Finally, some higher inductive types, like the truncation, can be added to CTT. We would like to understand how these interact with $\triangleright$.

Related work. Another type theory with a computational interpretation of functional extensionality, but without equality reflection, is observational type theory (OTT) [2]. We found CTT's prototype implementation, its presheaf semantics, and its interval as a tool for controlling unfoldings, most convenient for developing our combination with GDTT, but extending OTT similarly would provide an interesting comparison.

Spitters [25] used the interval of the internal logic of cubical sets to model identity types. Coquand [11] defined the composition operation internally to obtain a model of type theory. We have extended both these ideas to a full model of CTT. Recent independent work by Orton and Pitts [23] axiomatises a model for CTT without a universe, again building on Coquand [11]. With the exception of the absence of the universe, their development is more general than ours. Our semantic developments are sufficiently general to support the sound addition of guarded recursive types to CTT.

Acknowledgements. We gratefully acknowledge our discussions with Thierry Coquand, and the comments of our reviewers. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU). Aleš Bizjak was supported in part by a Microsoft Research PhD grant.

References

[1] Andreas Abel and Andrea Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *APLAS*, pages 140–158, 2014.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV*, pages 57–68, 2007.

[3] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208, 2013.

[4] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS*, pages 213–222, 2013.

[5] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *LMCS*, 8(4), 2012.

[6] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, pages 119–132, 2011.

[7] Aleš Bizjak and Rasmus Ejlers Møgelberg. A model of guarded recursion with clock synchronisation. In *MFPS*, pages 83–101, 2015.

[8] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *FoSSaCS*, pages 20–35, 2016.

[9] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *FoSSaCS*, pages 407–421, 2015.

[10] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. Unpublished, 2016.

[11] Thierry Coquand. Internal version of the uniform Kan filling condition. Unpublished, 2015. URL: `http://www.cse.chalmers.se/~coquand/shape.pdf`.

[12] Peter Dybjer. Internal type theory. In *TYPES '95*, pages 120–134, 1996.

[13] Martin Hofmann. *Extensional constructs in intensional type theory.* Springer, 1997.

[14] Martin Hofmann and Thomas Streicher. Lifting Grothendieck universes. Unpublished, 1999. URL: `http://www.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf`.

[15] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. *arXiv:1211.2851*, 2012.

[16] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic.* Springer, 1992.

[17] Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73*, pages 73–118, 1975.

[18] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[19] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Programming*, 18(1):1–13, 2008.

[20] Rasmus Ejlers Møgelberg. A type theory for productive coprogramming via guarded recursion. In *CSL-LICS*, 2014.

[21] Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

[22] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[23] Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. In *CSL*, 2016.

[24] Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, LFCS, University of Edinburgh, 1992.

[25] Bas Spitters. Cubical sets as a classifying topos. *TYPES*, 2015.

[26] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute for Advanced Study, 2013.

A zipWith Preserves Commutativity

We provide a formalisation of Sec. 2.4 which can be verified by our type checker. This file, among other examples, is available in the `gctt-examples` folder in the type-checker repository.

```
module zipWith_preserves_comm where

Id (A : U) (a0 a1 : A) : U = IdP (⟨i⟩ A) a0 a1
data nat = Z | S (n : nat)

-- Streams of natural numbers
StrF (S : ▷ U) : U = (n : nat) * ▷ [S' ← S] S'

Str : U = fix (StrF Str)

-- The canonical unfold lemma for Str
StrUnfoldPath : Id U Str (StrF (next Str))
 = ⟨i⟩ StrF (dfix U StrF [(i=1)])

unfoldStr (s : Str) : (n : nat) * ▷ Str
 = transport StrUnfoldPath s

foldStr (s : (n : nat) * ▷ Str) : Str
 = transport (⟨i⟩ StrUnfoldPath @ -i) s

cons (n : nat) (s : ▷ Str) : Str = foldStr (n, s)
head (s : Str) : nat = s.1
tail (s : Str) : ▷ Str = (unfoldStr s).2

-- Defining zipWith
zipWithF (f : nat → nat → nat) (rec : ▷ (Str → Str → Str))
 : Str → Str → Str
 = (λ (s1 s2 : Str) →
      (cons (f (head s1) (head s2))
            (next [zipWith' ← rec, s1' ← tail s1 , s2' ← tail s2]
                  zipWith' s1' s2')))

zipWith (f : nat → nat → nat) : Str → Str → Str
 = fix (zipWithF f zipWith)

zipWithUnfoldPath (f : nat → nat → nat)
 : Id (Str → Str → Str)
      (zipWith f)
      (zipWithF f (next (zipWith f)))
 = ⟨i⟩ zipWithF f (dfix (Str → Str → Str) (zipWithF f) [(i=1)])

-- Commutativity property
comm (f : nat → nat → nat) : U = (m n : nat) → Id nat (f m n) (f n m)

-- zipWith preserves commutativity.
zipWith_preserves_comm (f : nat → nat → nat) (c : comm f)
 : (s1 s2 : Str) → Id Str (zipWith f s1 s2) (zipWith f s2 s1)
 = fix
   (λ (s1 s2 : Str) →
      ⟨i⟩ comp (⟨_⟩ Str)
               (cons (c (head s1) (head s2) @ i)
                     (next [q ← zipWith_preserves_comm
                           ,t1 ← tail s1
                           ,t2 ← tail s2]
                           q t1 t2 @ i))
               [(i=0) → ⟨j⟩ zipWithUnfoldPath f @ -j s1 s2
               ,(i=1) → ⟨j⟩ zipWithUnfoldPath f @ -j s2 s1])
```

Chapter 5

Parametric quantifiers for dependent type theory

Parametric Quantifiers for Dependent Type Theory

ANDREAS NUYTS, KU Leuven, Belgium
ANDREA VEZZOSI, Chalmers University of Technology, Sweden
DOMINIQUE DEVRIESE, KU Leuven, Belgium

Polymorphic type systems such as System F enjoy the parametricity property: polymorphic functions cannot inspect their type argument and will therefore apply the same algorithm to any type they are instantiated on. This idea is formalized mathematically in Reynolds's theory of relational parametricity, which allows the metatheoretical derivation of parametricity theorems about all values of a given type. Although predicative System F embeds into dependent type systems such as Martin-Löf Type Theory (MLTT), parametricity does not carry over as easily. The identity extension lemma, which is crucial if we want to prove theorems involving equality, has only been shown to hold for small types, excluding the universe.

We attribute this to the fact that MLTT uses a single type former Π to generalize both the parametric quantifier $\forall$ and the type former $\to$ which is non-parametric in the sense that its elements may use their argument as a value. We equip MLTT with parametric quantifiers $\forall$ and $\exists$ alongside the existing Π and Σ, and provide relation type formers for proving parametricity theorems internally. We show internally the existence of initial algebras and final co-algebras of indexed functors both by Church encoding and, for a large class of functors, by using sized types.

We prove soundness of our type system by enhancing existing iterated reflexive graph (cubical set) models of dependently typed parametricity by distinguishing between edges that express relatedness of objects (bridges) and edges that express equality (paths). The parametric functions are those that map bridges to paths.

We implement an extension to the Agda proof assistant that type-checks proofs in our type system.

CCS Concepts: • **Software and its engineering → Polymorphism**; *Formal methods*; *Functional languages*; *Syntax*; *Semantics*; • **Theory of computation** → *Proof theory*;

Additional Key Words and Phrases: Parametricity, cubical type theory, presheaf semantics, sized types, Agda

ACM Reference Format:
Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric Quantifiers for Dependent Type Theory. *Proc. ACM Program. Lang.* 1, ICFP, Article 32 (September 2017), 29 pages.
https://doi.org/10.1145/3110276

1 INTRODUCTION

Many type systems and functional programming languages support functions that are parametrized by a type. For example, we may create a tree flattening function $\mathsf{flatten}\,\alpha : \mathsf{Tree}\,\alpha \to \mathsf{List}\,\alpha$ that works for any type α. If the implementation of a parametrized function does not inspect the particular type α that it is operating on, possibly because the type system prohibits this, then the function is said to be *parametric*: it applies the same algorithm to all types. From this knowledge, we obtain various useful 'free theorems' about the function [Reynolds 1983; Wadler 1989]. For example, if we have a function $f : A \to B$, then we know that $\mathsf{listmap}\,f \circ \mathsf{flatten}\,A = \mathsf{flatten}\,B \circ \mathsf{treemap}\,f$. If parametricity is enforced by the type system, as is the case in System F but also in a programming

2475-1421/2017/9-ART32
https://doi.org/10.1145/3110276

language like Haskell, then we can deduce such free theorems purely from a function's type signature, without knowledge of its implementation. This allows parts of a function's contract to be enforced by the type-checker; a powerful feature.

Existing work on parametricity in dependent type systems such as Martin-Löf Type Theory (MLTT) has been able to show that the expected parametricity results hold for functions that produce values of a small type [Atkey et al. 2014; Krishnaswami and Dreyer 2013; Takeuti 2001]. Below, we show with a simple example that in existing dependent type systems, parametricity theorems can break down where large types are involved. The central aim of this paper is to resolve this issue by equipping dependent type theory with additional parametric quantifiers.

Representation independence in System F. In order to expose the problem that occurs in dependent type theory, we will elaborate an example that shows the power of parametricity in System F, but which breaks down in dependent type theory. Assume that A is a type that is essentially an interface, listing the operations that its elements provide. Then typically, we will not directly construct values of type A; rather, we will construct representations of them in some type C, from which we can extract the operations using a function $r : C \to A$. A function f of type $\mathsf{RI}\ A\ B :\equiv \forall\gamma.(\gamma \to A) \to (\gamma \to B)$ for some fixed type B, is then a function that, for any type γ that implements the interface A as witnessed by $r : \gamma \to A$, produces a map $\gamma \to B$. Parametricity now asserts that f is representation independent: it can only use the argument $c : \gamma$ through its operations $r\ c : A$ and is thus oblivious to the particular implementation. Hence, elements of type $\mathsf{RI}\ A\ B$ are in one-to-one correspondence with functions $A \to B$.

Representation polymorphism in dependent type theory. Dependent type theory departs from System F in that it erases the strict dichotomy between types and values. The result is a system in which types can depend on values, and can appear as values themselves, possibly as computational content of other values (e.g. we can consider lists of types).

The function type former $\to$ from System F, is replaced with the type former Π (called the product type, dependent function type or simply Π-type) in dependent type theory. If S is a type and T is a type depending on a variable $x : S$, then the type $\Pi(x : S).T$ contains functions f that map any value $s : S$ to a value $f\,s : T[s/x]$. When T does not depend on x, we simply write $S \to T$ and have recovered the ordinary function type from System F.

If we disregard parametricity, we may also use Π to recover the $\forall$ type former from System F. If the domain S is some type of types $\mathcal{U}$, also called a universe, then the function type $\Pi(\alpha : \mathcal{U}).T$ corresponds to the polymorphic type $\forall\alpha.T$ from System F. So we can translate RI to dependent types as $\mathsf{RI}\ A\ B :\equiv \Pi(C : \mathcal{U}).(C \to A) \to (C \to B)$. However, representation independence is not enforced for this type, and an easy counterexample can be constructed if we let B be the universe $\mathcal{U}$ itself. Then we can break representation independence by directly leaking the implementation type C to the end user:

$$\mathsf{leak} = \lambda C.\lambda r.\lambda c.C : \Pi(C : \mathcal{U}).(C \to A) \to (C \to \mathcal{U}) \tag{1}$$

Wrapping up. We claim that while dependent type theory clearly takes a step forward from System F in that it allows any kind of dependencies, it takes a step back by unifying $\forall$ and $\to$ in a single type constructor. The problem is that functions $f : \forall\alpha.T$ and $g : P \to Q$ differ not only in that f is dependent and takes a type as argument whereas g is non-dependent and takes a value as argument; they also differ in that f is parametric and uses its argument solely for type-checking purposes, whereas g is non-parametric and is allowed to use its argument as a value. It is the second property of $\forall$ that produces the free theorems we want.

In order to restore parametricity for large types in dependent type theory, we reinstate the parametric quantifier $\forall$ from System F alongside the non-parametric quantifier Π (also $\to$) in dependent type theory. The type formation rules for both quantifiers have the exact same premises.

This means that we can quantify parametrically or not over either type-like or value-like arguments, making the distinction between parametricity and non-parametricity orthogonal to the distinction between type-level and value-level arguments, which we seek to erase in dependent type theory. This leads to four situations of which only two existed (at the value level) in System F.

(1) As in System F, we can quantify parametrically over a type argument. An example is the (proper) Church encoding of lists: $\mathsf{ChList}\, B = \forall(X : \mathcal{U}).¶X \to ¶(B \to X \to X) \to X$ (the ¶ modality is explained later).
(2) Unlike in System F, we can quantify non-parametrically over a type argument. A (non-dependent) example is the cons constructor of $\mathsf{List}\,\mathcal{U}$, which has the type $\mathsf{cons} : \mathcal{U} \to \mathsf{List}\,\mathcal{U} \to \mathsf{List}\,\mathcal{U}$. Clearly, this function uses its first argument not just for type checking purposes. Rather, the *value* X can be retrieved from the list $\mathsf{cons}\, X\, Xs$ using the list recursor. Another example is the function type former $\lambda X.\lambda Y.(X \to Y) : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$. The arguments X and Y are not used just for type-checking (in fact they do not even occur in the type $\mathcal{U} \to \mathcal{U} \to \mathcal{U}$); rather, they determine the output *value* $X \to Y$. Non-parametric *dependent* quantification over a type variable is also common. For example, algebraic structures can be represented as dependent tuples and these must be non-parametric in their underlying type, lest we identify structures as soon as there is a homomorphism between them.
(3) Unlike in System F, we can quantify parametrically over a value argument. We have a type Size that is similar to the natural numbers but enforces a form of well-behavedness for functions that have Size as their domain. Let $\mathsf{Vec}_i\, A$ be the type of vectors of length $i : \mathsf{Size}$ over A, where vectors of different lengths are considered equal if the truncation of the longer one is equal to the shorter one. Now consider the type $\forall(i : \mathsf{Size}).\mathsf{Vec}_i\, A$. Parametricity ensures that a function of this type will produce equal (i.e. compatible) vectors for all sizes. In other words, this is the type of infinite streams of elements of A.
(4) Like in System F, we can quantify non-parametrically over a value argument. Any ordinary term-level function from System F is an example of this. A more intriguing example is the function $\lambda i.\mathsf{Vec}_i\, A : \mathsf{Size} \to \mathcal{U}$ which maps sizes to types, but also provides a notion of heterogeneous (cross-type) equality between elements of $\mathsf{Vec}_i\, A$ and $\mathsf{Vec}_j\, A$.

Although up until this point, we appealed to the intuition of parametric functions as 'not inspecting an argument', this intuition diverges from the relational formulation when we consider parametric quantification over a data type. Relational parametricity asserts that related inputs will lead to related outputs. The **identity extension lemma** (IEL) moreover implies that relatedness in a closed type, means equality. No assertions are made, however, about unrelated inputs. If we have a type Nat in which (unlike in Size) different natural numbers are considered unrelated, then we can allow to pattern match on them. However, a function $T : \mathsf{Nat} \to \mathcal{U}$ may then not provide a notion of equality between $T\, m$ and $T\, n$ and a function of type $\forall(n : \mathsf{Nat}).T\, n$ need not produce equal output for different numbers (as we even lack the notion of equality). This is a situation that does not apply in basic System F (or Fω), where any two elements of a given kind, are related.

Contributions.

(1) We present a dependent type system **ParamDTT** in which dependencies can be either *parametric* (♯) or *continuous* (**id**). Correspondingly, we obtain (predicative) relationally parametric quantifiers $\forall$ and $\exists$ alongside the usual continuous quantifiers Π and Σ.
(2) We make parametricity theorems provable internally using a type former called Glue (first used by Cohen et al. [2016] in order to achieve computational univalence), and its (novel) dual which we call Weld. These are an alternative for the operators by Bernardy et al. [2015]. Both Glue and Weld have some dependencies that are not continuous and that we cannot prove further parametricity theorems about. This is represented by a third *pointwise* modality

(¶). As such, these type formers cannot be self-applied and iterated parametricity is not fully available internally. We reframe and internalize IEL in the form of an axiom called the *path degeneracy axiom*, enabling us to prove parametricity theorems involving equality.

(3) We construct Church initial algebras and final co-algebras of indexed functors, showing that indexed (co)-recursive types can be built up from simpler components. We prove their universal properties (up to universe level issues) internally, which to our knowledge has not been done before in any type system. These internal proofs have some pointwise dependencies, indicating that internal parametricity does not apply again to those dependencies.

(4) Annotating (co)-recursive types with a size bound on their elements is a modular way to enforce termination and productivity of programs. This is a use case for parametric quantification over values, as we do not want to view an object's size bound as computational content of the object. We construct initial algebras and final co-algebras of a large class of indexed functors using induction on, and parametric quantification over size bounds. We again prove their universal properties internally.

(5) We implement an extension to the dependently typed language Agda, which type-checks ParamDTT and thus shows that its computational behaviour is sufficiently well-behaved to allow for automated type-checking. [1] We expect that ParamDTT minus its equality axioms, which block computation of the J-rule, satisfies all desired computational properties.

(6) We prove the soundness of the type system by constructing a presheaf model in terms of iterated reflexive graphs (more commonly called cubical sets), based on the reflexive graph model by Atkey et al. [2014] and enhancements by Bernardy et al. [2015]. An important innovation in our model is that our iterated graphs have two flavours of edges: bridges express relatedness, and paths express heterogeneous equality of objects living in related types. If we were to model parametricity of System F in the same model, we would use bridges to represent relatedness of types, and paths to represent relatedness of terms. Correspondingly, continuous functions are those that respect edge flavours, whereas parametric functions are those that strengthen bridges to paths.

Overview. In Section 2, we give an informal overview of ParamDTT and its features. In Section 3, we present the formal typing rules and relate the system to MLTT and predicative System Fω. In Section 4, we treat Church encoding and sized types. In Section 5, we give an overview of the presheaf model that proves soundness of ParamDTT. A more complete treatment of the model is found in [Nuyts 2017]. We conclude in Section 6 with a discussion of related work and future research directions.

2 A PROGRAMMER'S PERSPECTIVE

Before we show the formal rules of our type system, we present the system from a programmer's perspective. We consider the typical but simple example type of polymorphic identity functions: $\forall(X : \mathcal{U}).X \to X$.

Modalities. The $\forall$ quantifier is syntactic sugar for $\Pi^\sharp(X : \mathcal{U}).X \to X$: a Π-type annotated with the parametric modality $\sharp$. We can construct a value of this type by annotating a lambda with the same modality: $\lambda(X^\sharp : \mathcal{U}).\sqcup$. In the body, the variable X is available in the context, annotated as $X^\sharp : \mathcal{U}$ to remind us that it should only be used in parametric positions (which we color magenta as a guide to the reader). A variable that is in the context parametrically ($X^\sharp : \mathcal{U}$) does not type-check as a value of type $\mathcal{U}$. Luckily, when we next use a normal lambda $\lambda(x : X).\sqcup$ (i.e. annotated with the continuous modality **id**) to construct a value of type $X \to X$, the type annotation X for x is a parametric position. As such, it is not type-checked in the current context ($X^\sharp : \mathcal{U}$), but in

[1] Available in the artifact, and on Github as the 'parametric' branch of Agda: https://github.com/agda/agda/tree/parametric

the context $(X : \mathcal{U})$, where all parametric variables have been rendered continuous. This context modification is a common theme in modal typing rules: ♯-modal subterms like our X are checked in modified contexts, defined by formally *left-dividing* the current context $(X^\sharp : \mathcal{U})$ by the ♯-modality: $(\sharp \setminus (X^\sharp : \mathcal{U})) = (X : \mathcal{U})$. In the body of the second lambda, we simply return variable x of type X to finish our example.

Another parametric position is the argument that we pass to a parametric function. For example, with $f : \forall(X : \mathcal{U}).X \to X$ in the context, we can construct another value of the same type by composing f with itself: $\lambda(X^\sharp : \mathcal{U}).\lambda(x : X).f\ X^\sharp\ (f\ X^\sharp\ x)$. The variable X is used (twice) as the argument to a parametric function, and as such, it is not type-checked in context $\Gamma = (X^\sharp : \mathcal{U}, x : X)$ (where it would not be accepted), but in $(\sharp \setminus \Gamma) = (X : \mathcal{U}, x : X)$.

As a guide to the reader, we color subterms and variable bindings according to their modality: magenta for parametricity (♯), black for continuity (**id**), blue for the pointwise modality (denoted ¶, see further below), and orange for any unknown modality (some typing rules work for an arbitrary modality μ). Because we want our language to be unambiguous even without color, we will sometimes additionally insert a modality symbol to disambiguate. Continuity (**id**) is considered the default and will be omitted.

Internal parametricity: paths and bridges. A compelling feature of our type system is that we have internal parametricity: free theorems about parametric functions can be derived internally. Imagine that we have a function $f : \forall(X : \mathcal{U}).X \to X$, a type $X^\sharp : \mathcal{U}$ and a value $x : X$ and we want to use parametricity of f to prove that $f\ X^\sharp\ x$ is equal to x. We can do this inside the language, using essentially the same approach that one would take when using binary relational parametricity of System F. There, for every $x : X$, we would construct a relation R_x between the unit type $\top$ and X such that $R_x(u, y)$ iff $x = y$. Since f, being parametric in its first argument, maps R_x-associated second arguments to R_x-associated output, we have that $R_x(f\ \top^\sharp\ \mathrm{tt}, f\ X^\sharp\ x)$, i.e. $f\ X^\sharp\ x = x$.

To construct the relation R_x inside the language, we will construct a *bridge* from $\top$ to X. Such a bridge can best be thought of as a line connecting two values, possibly living in different types. The precise meaning of a bridge depends on the types concerned; a bridge from $B_0 : \mathcal{U}$ to $B_1 : \mathcal{U}$ gives meaning to statements of the form '$b_0 : B_0$ and $b_1 : B_1$ are related' (expressed by a heterogeneous (cross-type) bridge from b_0 to b_1) or '$p_0 : B_0$ and $p_1 : B_1$ are equal' (expressed by a *path* from p_0 to p_1). Note that, unlike in existing accounts of relational parametricity, we distinguish between relatedness and (possibly heterogeneous) equality.

A proof of $R_x(u, y)$ will be represented internally as a *path* from $u : \top$ to $y : X$. A path between values p_0 and p_1 can also be thought of as a line connecting these values. Just like bridges, paths may be heterogeneous, but when they are homogeneous (i.e. when they stay within a single type), they are necessarily constant, implying equality of their endpoints. Moreover, paths are respected by all functions. These properties make the path relation a good notion of heterogeneous equality: a congruence that reduces to equality whenever equality is meaningful. So in order to prove that $f\ X^\sharp\ x = x$, it will be sufficient to construct a path from $f\ X^\sharp\ x$ to x.

We internalize both bridges and paths using a special pseudo-type $\mathbb{I}$ called the interval, which consists of two elements 0 and 1 connected by a bridge. Since continuous functions, by definition in the model, respect bridges, a bridge from B_0 to B_1 can be represented as a function $B : \mathbb{I} \to \mathcal{U}$ such that $B\,0 \equiv B_0$ and $B\,1 \equiv B_1$ definitionally. Since parametric functions, by definition in the model, strengthen bridges to paths, a path from $p_0 : B_0$ to $p_1 : B_1$ can be represented as a function $p : \forall(i : \mathbb{I}).B\,i$ such that $p\,0^\sharp \equiv p_0$ and $p\,1^\sharp \equiv p_1$. Meanwhile, heterogeneous bridges take the form $b : \Pi(i : \mathbb{I}).B\,i$. The typing rules make sure that the types $\forall(x : A).T\,x$ and $\Pi(x : A).T\,x$ can be formed precisely when we have a continuous function $T : A \to \mathcal{U}$; when $\mathbb{I}$ is the domain, this says that there needs to be a bridge between the types before we can consider bridges or paths

between their values. An internal *path degeneracy axiom* degax finally asserts that non-dependent (homogeneous) paths are in fact constant.

Turning a function into a bridge. Note that the relation R_x we used in the System F proof, is in fact the graph of the function $\lambda u.x : \top \to X$. As mentioned above, we intend to internalize R_x as a bridge from $\top$ to X, i.e. a function $/\lambda u.x\backslash : \mathbb{I} \to \mathcal{U}$ such that $/\lambda u.x\backslash 0$ reduces to $\top$ and $/\lambda u.x\backslash 1$ reduces to X. In fact, an operator $/\sqcup\backslash$ that turns a function into the bridge representing its graph relation, can be implemented using either the primitive Glue type former, or its dual called Weld, which we introduce in Section 3.2. For now, we just assume that we have a bridge $/\lambda u.x\backslash$ and that it comes with a function $\mathsf{push} : \forall(i : \mathbb{I}).\top \to /\lambda u.x\backslash i$ from the domain, such that $\mathsf{push}\, 0^\sharp \equiv \mathsf{id}_\top : \top \to \top$ and $\mathsf{push}\, 1^\sharp \equiv \lambda u.x : \top \to X$; and a function $\mathsf{pull} : \forall(i : \mathbb{I})./\lambda u.x\backslash i \to X$ to the codomain, such that $\mathsf{pull}\, 0^\sharp \equiv \lambda u.x : \top \to X$ and $\mathsf{pull}\, 1^\sharp \equiv \mathsf{id}_X : X \to X$. Now consider the following composite:

$$\mathbf{1} \xrightarrow{\mathsf{push}\, i^\sharp} /\lambda u.x\backslash i \xrightarrow{f\,(/\lambda u.x\backslash i)^\sharp} /\lambda u.x\backslash i \xrightarrow{\mathsf{pull}\, i^\sharp} X.$$

For $i \equiv 0$, it reduces to the constant function $\lambda u.x$, while for $i \equiv 1$, it reduces to $\lambda u.f\, X^\sharp\, x$. Thus, applying this to $\mathsf{tt} : \top$, we obtain a homogeneous (non-dependent) path

$$p :\equiv \lambda i^\sharp.\, (\mathsf{pull}\, i^\sharp \circ f\,(/\lambda u.x\backslash i)^\sharp \circ \mathsf{push}\, i^\sharp)\, \mathsf{tt} : \sharp\mathbb{I} \to X, \qquad p\, 0^\sharp \equiv x, \qquad p\, 1^\sharp \equiv f\, X^\sharp\, x.$$

Finally, since this is a non-dependent (homogeneous) path, the aforementioned path degeneracy axiom asserts that it is constant, implying that $x =_X f\, X^\sharp\, x$.

Before we proceed: the pointwise modality. One important aspect of our system has been tucked under the carpet in the above example: the Glue and Weld type formers, as well as the graph type former $/\sqcup\backslash$ implemented in terms of either of them, break the relational structure. This is reflected syntactically by a third *pointwise* modality (¶), which annotates dependencies that have no action on bridges. So to be precise, the above example does not show that any function of type $\forall(X : \mathcal{U}).X \to X$ is the identity; rather, every such function can be weakened to the type $\forall(X : \mathcal{U}).¶X \to X$ (forgetting its action on bridges in the second argument) and we have proven that all functions of *that* type are the identity. In practice, this means the proof is perfectly usable and valid, but we cannot apply another parametricity argument to the proof term. This restriction is a limitation of our current model, but an acceptable one, as we will argue in what follows.

3 THE TYPE SYSTEM, FORMALLY

With the general ideas of ParamDTT established, this section presents the system formally. The first part treats the core type system, which is just MLTT with modality annotations. The second part explains the machinery we use for internal parametricity. The third part adds two types Nat and Size of natural numbers, and we conclude in the fourth part by embedding MLTT and predicative System Fω in our system in two ways.

3.1 Core Typing Rules: Annotating Martin-Löf Type Theory

Modalities. The judgements and typing rules of our system are similar to MLTT, except that, as discussed, every dependency is equipped with a modality: either pointwise (¶), continuous (**id**) or parametric (♯). We follow the general approach developed by Pfenning [2001] and Abel [2006, 2008]. A dependency's modality expresses how it acts on bridges and paths. Functions of all three modalities respect paths. Continuous functions moreover respect bridges, while parametric functions strengthen them to paths, and pointwise functions do not act on bridges whatsoever. Every dependency can be viewed as pointwise by ignoring its action on bridges. Because equal values are also related, our model allows to weaken paths to bridges; this weakening allows to view parametric dependencies as continuous. We express these findings as an order relation on modalities (Fig. 1).

If a term t_1 depends μ_1-modally on a variable x and t_2 depends μ_2-modally on y, then the term $t_2[t_1/y]$ depends on x under a composed modality $\mu_2 \circ \mu_1$. This composition of modalities is defined by the first table in Fig. 1, and follows immediately from the action on bridges and paths. E.g. $\mu \circ \P = \P$, because if the inner function forgets bridges, then the outer one cannot retrieve them, and $\mu \circ \sharp = \sharp$ because if the inner function strengthens bridges to paths, then the outer function has to respect those paths. Note that composition preserves order in both operands.

$\P \le \mathbf{id} \le \sharp$

$\downarrow \circ \rightarrow$	$\P$	$\mathbf{id}$	$\sharp$
$\P$	$\P$	$\P$	$\sharp$
$\mathbf{id}$	$\P$	$\mathbf{id}$	$\sharp$
$\sharp$	$\P$	$\sharp$	$\sharp$

$\downarrow \setminus \rightarrow$	$\P$	$\mathbf{id}$	$\sharp$
$\P$	$\P$	$\sharp$	$\sharp$
$\mathbf{id}$	$\P$	$\mathbf{id}$	$\sharp$
$\sharp$	$\P$	$\mathbf{id}$	$\mathbf{id}$

Fig. 1. Composition, left division and order of modalities.

Formally, modalities come into play wherever dependencies appear in the type system. First of all, the term on the right of a judgement depends on the context variables, so we annotate those with modalities: we write $x : A$ for continuous, $x^\sharp : A$ for parametric and $x^\P : A$ for pointwise variables. Secondly, a term in the conclusion of an inference rule, also depends on the terms $t : T$ in the premises. As explained before, such terms may be in μ-modal position, in which case the premise will not be $\Gamma \vdash t : T$ but rather $\mu \setminus \Gamma \vdash t : T$, which can be read as $\Gamma \vdash t^\mu : T$ (a non-existent judgement form). This left-division of context Γ by modality μ replaces every dependency $x^\nu : A$ by $x^{\mu\setminus\nu} : A$, defined by the second table in Fig. 1. The left division selects the least modality $\mu \setminus \nu$ such that $\nu \le \mu \circ (\mu \setminus \nu)$. In other words, $\mu \setminus \nu \le \rho \Leftrightarrow \nu \le \mu \circ \rho$ for all ρ, i.e. left division by μ is left adjoint to postcomposition with μ. In general, we take care to maintain admissibility of the following structural rules:

$$\frac{\Gamma, x^\mu : T, \Delta \vdash J \qquad \mu \setminus \Gamma \vdash t : T}{\Gamma, \Delta[t/x] \vdash J[t/x]}\text{subst} \qquad \frac{\Gamma, x^\mu : T, \Delta \vdash J \qquad \nu \le \mu}{\Gamma, x^\nu : T, \Delta \vdash J}\text{mod-wkn}$$

With this modality machinery in place, we can now discuss our typing rules.

Contexts. Contexts are formed by starting from the empty context (c-em) and adding variables in modalities of your choice (c-ext). Context variables are valid terms if their modality is continuous or less (t-var).

$$\frac{}{\vdash \mathrm{Ctx}}\text{c-em} \qquad \frac{\Gamma \vdash T\ \mathrm{type}}{\Gamma, x^\mu : T \vdash \mathrm{Ctx}}\text{c-ext} \qquad \frac{\Gamma \vdash \mathrm{Ctx} \quad (x^\mu : T) \in \Gamma \quad \mu \le \mathbf{id}}{\Gamma \vdash x : T}\text{t-var}$$

Universes. There is a countable hierarchy of universes, each one living in the next (t-Uni). Elements of a universe can be coerced to higher universes (t-lift). Elements of the universe can be turned into types (ty), and this operation is parametric because it shifts T to the type level, preventing any further use as a value.

$$\frac{\Gamma \vdash \mathrm{Ctx} \qquad \ell \in \mathbb{N}}{\Gamma \vdash \mathcal{U}_\ell : \mathcal{U}_{\ell+1}}\text{t-Uni}, \qquad \frac{\Gamma \vdash T : \mathcal{U}_k \qquad k \le \ell \in \mathbb{N}}{\Gamma \vdash T : \mathcal{U}_\ell}\text{t-lift}, \qquad \frac{\sharp \setminus \Gamma \vdash T : \mathcal{U}_\ell}{\Gamma \vdash T\ \mathrm{type}}\text{ty},$$

Definitional equality. There are equality judgements $\Gamma \vdash s \equiv t : T$ and $\Gamma \vdash S \equiv T\ \mathrm{type}$. We omit the rules that make definitional equality a congruence and an equivalence relation. The conversion rule (t-conv) allows to convert terms between equal types.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B\ \mathrm{type}}{\Gamma \vdash a : B}\text{t-conv},$$

Quantification. We have universal and existential quantifiers for every modality (t-Π, t-Σ). We denote them as Π^μ and Σ^μ for general modalities, but we abbreviate the continuous ones as Π and Σ and the parametric ones as $\forall$ and $\exists$. Moreover, in the non-dependent case we will write $\mu A \to B$ and $\mu A \times B$. Quantified types are continuous in both domain A and codomain B. Indeed, A and B are not just provided for type-checking but they *determine* the output value $\mathrm{Q}(x : A).B$. The variance of B's dependency on x is worth a mention: we want $b : B$ to be meaningful if an element $a^\mu : A$ is given μ-modally. Since in the claim $b : B$, the type B is in a parametric position, it needs to be

defined for $a^{\sharp\setminus\mu} : A$. For $\mu = \mathbf{id}$ or $\mu = \sharp$, we have that $\sharp \setminus \mu$ is $\mathbf{id}$, so B depends continuously on x. However, if $\mu = ¶$, then the codomain may be pointwise in x, since $\sharp \setminus ¶ = ¶$.

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell \qquad \Gamma, x^{\sharp\setminus\mu} : A \vdash B : \mathcal{U}_\ell}{\Gamma \vdash \Pi^\mu(x : A).B : \mathcal{U}_\ell}\text{t-}\Pi \qquad \frac{\Gamma \vdash A : \mathcal{U}_\ell \qquad \Gamma, x^{\sharp\setminus\mu} : A \vdash B : \mathcal{U}_\ell}{\Gamma \vdash \Sigma^\mu(x : A).B : \mathcal{U}_\ell}\text{t-}\Sigma$$

Functions. μ-Modal functions are created using λ-abstraction over μ-modal values (t-λ). μ-Modal function applications depend μ-modally on the argument provided (t-ap). We omit β- and η-rules for functions.

$$\frac{\Gamma, x^\mu : A \vdash b : B}{\Gamma \vdash \lambda(x^\mu : A).b : \Pi^\mu(x : A).B}\text{t-}\lambda \qquad \frac{\Gamma \vdash f : \Pi^\mu(x : A).B \qquad \mu \setminus \Gamma \vdash a : A}{\Gamma \vdash f\ a^\mu : B[a/x]}\text{t-ap}$$

Pairs. The μ-modal existential type $\Sigma^\mu(x : A).B$ contains pairs (a^μ, b) where $a^\mu : A$ and $b : B[a/x]$ (t-pair). We have a dependent eliminator for ν-modal use of pairs (t-indpair), for which we omit the β-rule. Note that the non-dependent specialization of $\mathsf{ind}^{\mathbf{id}}_\exists$ corresponds to the 'unpack' eliminator for existentials in System F [see e.g. Pierce 2002, ch. 24].

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma^\mu(x : A).B\ \mathsf{type}\\ \mu \setminus \Gamma \vdash a : A\\ \Gamma \vdash b : B[a/x]\end{array}}{\Gamma \vdash (a^\mu, b) : \Sigma^\mu(x : A).B}\text{t-pair} \qquad \frac{\begin{array}{l}\Gamma, z^\nu : \Sigma^\mu(x : A).B \vdash C\ \mathsf{type}\\ \Gamma, x^{\nu\circ\mu} : A, y^\nu : B \vdash c : C[(x^\mu, y)/z]\\ \nu \setminus \Gamma \vdash p : \Sigma^\mu(x : A).B\end{array}}{\Gamma \vdash \mathsf{ind}^\nu_{\Sigma^\mu}(z.C, x.y.c, p) : C[p/z]}\text{t-indpair}$$

Using $\mathsf{ind}^{\mathbf{id}}_\Sigma$, we can implement continuous projections for $\Sigma(x : A).B$ as

$$\mathsf{fst} : \Sigma(x : A).B \to A \qquad \mathsf{snd} : \Pi(p : \Sigma(x : A).B).B[\mathsf{fst}\, p/x]$$

$$\mathsf{fst}\, p = \mathsf{ind}^{\mathbf{id}}_\Sigma(z.A, x.y.x, p), \qquad \mathsf{snd}\, p = \mathsf{ind}^{\mathbf{id}}_\Sigma(z.B[\mathsf{fst}\, z/x], x.y.y, p).$$

The model supports the η-rule for $\mathsf{ind}^\nu_{\Sigma^\mu}$, allowing us to assume the η-rule $p \equiv (\mathsf{fst}\, p, \mathsf{snd}\, p)$. For $\Sigma^¶(x : A).B$, we can similarly implement a parametric first and a continuous second projection:

$$\mathsf{fst}^¶ : \sharp(\Sigma^¶(x : A).B) \to A \qquad \mathsf{snd}^¶ : \Pi\Big(p : \Sigma^¶(x : A).B\Big).B[\mathsf{fst}\, p/x]$$

$$\mathsf{fst}^¶\, p = \mathsf{ind}^\sharp_{\Sigma^¶}(z.A, x.y.x, p), \qquad \mathsf{snd}^¶\, p = \mathsf{ind}^{\mathbf{id}}_{\Sigma^¶}(z.A, x.y.y, p).$$

Since p depends on its first component pointwise and $\sharp \circ ¶ = ¶$, the term $\mathsf{fst}^¶\, p$ gets pointwise access, hence continuous access, to the first component of p. Again, we assume the η-rule for these projections.

Example 3.1. Consider the Church encoding of streams: $\mathsf{ChStr}\, A = \exists(X : \mathcal{U}_\ell).((X \to A \times X) \times X)$. The above rules allow us to build head and tail functions for this type, left as an exercise to the reader.

Identity types. We have an identity type $a =_A b$, continuous in A, a and b (t-Id).[2] The reflexivity constructor is parametric (t-refl). We can use an equality proof e with modality ν using the dependent eliminator J^ν.[3] We omit the β-rule. Our model supports the reflection rule (t=-rflct), which we will often not want to include as it breaks decidability of type-checking. Instead, we can take some of its consequences as axioms, such as function extensionality. A program that applies the J-rule to an instance of this axiom, will block. The model also supports definitional uniqueness of identity

[2] It may be surprising that the identity type $a =_A b$ is not parametric in the type A, as one might think it is only there for type-checking. However, if this were the case, then by parametricity, the type former $\sqcup =_\sqcup \sqcup$, of type $\forall(X : \mathcal{U}_\ell).X \to X \to \mathcal{U}_\ell$ would have to be constant (this claim holds pointwise in a and b). As such, the type $a =_A b$ would have to remain unchanged if we were to replace the type A, for example, with the related type $\top$ and $a, b : A$ with the heterogeneously equal values $\mathsf{tt}, \mathsf{tt} : \top$.

[3] A J-eliminator for every modality is actually a bit overkill, since the J-rule for a lesser modality follows from the J-rule for a greater one.

proofs (t=-UIP); which could be added to the type system either as-is, or in the form of special-case propositional or definitional rules (e.g. proofs of $a =_A a$ reduce to $\mathsf{refl}\, a$).

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell \qquad \Gamma \vdash a, b : A}{\Gamma \vdash a =_A b : \mathcal{U}_\ell}\text{t-Id} \qquad \frac{\sharp \setminus \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}\, a : a =_A a}\text{t-refl}$$

$$\frac{\begin{array}{ll}\sharp \setminus \Gamma \vdash a, b : A & \Gamma, y^\sharp : A, w^\nu : a =_A y \vdash C\ \mathsf{type} \\ \nu \setminus \Gamma \vdash e : a =_A b & \Gamma \vdash c : C[a/y, \mathsf{refl}\, a/w]\end{array}}{\Gamma \vdash \mathsf{J}^\nu(a, b, y.w.C, e, c) : C[b/y, e/w]}\text{t-J} \qquad \frac{\begin{array}{l}\sharp \setminus \Gamma \vdash f, g : \Pi^\mu(x : A).B \\ \Gamma \vdash p : \Pi^\mu(x : A).f\, x^\mu =_B g\, x^\mu\end{array}}{\Gamma \vdash \mathsf{funext}^\mu\, p : f =_{\Pi^\mu(x:A).B} g}$$

$$\left(\frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash e : a =_A b}{\Gamma \vdash a \equiv b : A}\text{t=-rflct}\right) \qquad \left(\frac{\Gamma \vdash e, e' : a =_A b}{\Gamma \vdash e \equiv e' : a =_A b}\text{t=-UIP}\right)$$

Example 3.2. The continuous J-rule allows us to prove transport: $\forall(X, Y : \mathcal{U}_\ell).(X =_{\mathcal{U}_\ell} Y) \to X \to Y$. Plugging in reflexivity, we get a term of type $\forall(X : \mathcal{U}_\ell).X \to X$. Parametricity will allow us to conclude, solely from the type, that this term is (pointwise) the identity function.

3.2 Internal Parametricity: Glueing and Welding

With the core typing rules established, let us now turn to the operators that will allow us to prove parametricity theorems internally. In Section 2, we used the type former $/\_\setminus$ for turning a function $f^\P : C \to D$ into a bridge $/f\setminus : \mathbb{I} \to \mathcal{U}$ from C to D. As mentioned, this type former is not a primitive; rather, it can be implemented in terms of either of the primitive type formers Glue and Weld, which we introduce in this section. Because they require a bit of machinery, we start the section with an introductory example in which we implement $/\_\setminus$ in terms of both Glue and Weld. The Glue type former was originally introduced with one additional prerequisite by Cohen et al. [2016] in a type system without modalities.

3.2.1 Introduction: Turning a Bridge into a Function. Let us take an $f^\P : C \to D$ and reiterate from Section 2 the properties that we need the type $/f\setminus : \mathbb{I} \to \mathcal{U}$ to satisfy. The type is a bridge between C and D, so we want to have that $/f\setminus 0 \equiv C$ and $/f\setminus 1 \equiv D$. Additionally, we want there to be a function $\mathsf{push} : \forall(i : \mathbb{I}).C \to /f\setminus i$ such that $\mathsf{push}\, 0^\sharp \equiv \mathsf{id}_C : C \to C$ and $\mathsf{push}\, 1^\sharp \equiv f : C \to D$. Finally, we also need $\mathsf{pull} : \forall(i : \mathbb{I})./f\setminus i \to D$ to the codomain, such that $\mathsf{pull}\, 0^\sharp \equiv f : C \to D$ and $\mathsf{pull}\, 1^\sharp \equiv \mathsf{id}_D : D \to D$. The property $\mathsf{pull}\, i^\sharp \circ \mathsf{push}\, i^\sharp \equiv f$ holds if i equals 0 or 1 and we want it to hold on the entire interval.

A construct called *systems* allows us to construct partially defined terms. For example, we can define $T :\equiv (i \doteq 0\,?\,C \mid i \doteq 1\,?\,D)$, a type that reduces to C if $i \doteq 0$ and to D if $i \doteq 1$. For general i, it is not defined; it only makes sense when the predicate $P :\equiv (i \doteq 0 \vee i \doteq 1)$ holds. Clearly, $/f\setminus i$ should extend T, in the sense that both types should be equal whenever T is defined. A naive solution would be to add a default clause, i.e. something like:

$$/f\setminus i :\equiv (i \doteq 0\,?\,C \mid i \doteq 1\,?\,D \mid \mathsf{else}\ E) \qquad \text{(not ParamDTT syntax).}$$

This approach is not quite right: it would allow us to relate any two types C and D by adding an arbitrary third type as a default clause and it is all but clear what that would mean. But let's consider it anyway. What would be the meaning of a path $p : \forall(i : \mathbb{I})./f\setminus i$ with this definition? The endpoints of the path would live in C and D, while the rest of the path lives in E. So we need some condition to decide whether elements $c : C$ and $d : D$ qualify as endpoints of a path in E (which normally has endpoints also in E). This condition could arise from a relation between C and $E[0/i]$ and one between D and $E[1/i]$, or in short a relation between T and E defined only when P holds. This is essentially how Glue and Weld work; however, they do not allow any kind of relation. The Weld type former takes a partially defined function $g^\P : E \to T$ whereas Glue takes $h^\P : T \to E$. The resulting types are denoted

$$\mathsf{Weld}\{E \to (P\,?\,T, g)\}, \qquad \mathsf{Glue}\{E \leftarrow (P\,?\,T, h)\},$$

and reduce to T when P is true. When P is false, they will be isomorphic to E.[4]

Using the Weld *operation.* In order to form $/f\backslash\, i$ using Weld, we need a diagram of the form $C \leftarrow E \to D$ that somehow encodes the graph of the function f.[5] In general, a relation can be represented by such a diagram if we let E be the type of related pairs, which for functions is isomorphic to the domain. So we set $E :\equiv C$ (Fig. 2, left column) and $g :\equiv (i \doteq 0\,?\,\mathsf{id}_C \mid i \doteq 1\,?\,f) : C \to T$ (Fig. 2, full arrows from left to middle column). With some syntactic sugar to avoid repeating the same predicate, we can then write (Fig. 2, middle column)

$$/f\backslash\, i :\equiv \mathsf{Weld}\{C \to (i \doteq 0\,?\,C, \mathsf{id}_C \mid i \doteq 1\,?\,D, f)\}, \qquad /f\backslash\, 0 \equiv C, \qquad /f\backslash\, 1 \equiv D.$$

Thinking of $/f\backslash\, i$ as a system with a default clause, it is clear how we can obtain elements of it: if P holds, we can take elements of T. If it does not hold, then we are (intuitively) in the default case and we can use a constructor-like function $\mathsf{push}\,i^\sharp :\equiv \mathsf{weld}\,(P\,?\,g) \equiv \mathsf{weld}\,(i \doteq 0\,?\,\mathsf{id}_C \mid i \doteq 1\,?\,f) : C \to /f\backslash\, i$ (Fig. 2, dashed arrow). Moreover, this function extends to the case where P *does* hold, and then it specializes to g, i.e. $\mathsf{push}\,0^\sharp \equiv \mathsf{id}_C$ and $\mathsf{push}\,1^\sharp \equiv f$. This internalizes the idea that the paths in the welded type come from the default case, while their endpoints are associated to them by the function g. Since $\mathsf{weld}\,(P\,?\,g)$ is meaningful regardless of whether P is true or false, it is a total function extending g. To summarize: given a partial type T, a total type E and a partially defined $g : E \to T$, welding extends T to a total type and g to a total function which takes the role of a constructor.

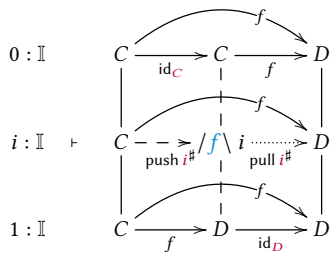

Fig. 2. Formation and elimination of $/f\backslash$ using Weld. The middle row depends on i and reduces to the top/bottom row when i equals 0/1. Welding produces $/f\backslash\, i$ and the dashed lines.

Finally, in order to construct $\mathsf{pull} : \forall(i : \mathbb{I}).\, /f\backslash\, i \to D$, we use the eliminator $\mathsf{ind}_{\mathsf{Weld}}$. It allows us to eliminate a value $w : /f\backslash\, i$ into a goal type D (which could in general depend on w) by inspecting how w was obtained: we need to handle elements created using $\mathsf{weld}\,(P\,?\,g)$ (Fig. 2, curved arrows) and, in the event that P holds, elements living in T (Fig. 2, full arrows from middle to right column). These cases are not disjoint: if P is true, then $\mathsf{weld}\,(P\,?\,g)$ creates elements of type T, so we need to handle them compatibly (Fig. 2, commutation of diagrams in top and bottom row). Thus, we can define $\mathsf{pull}\,i^\sharp$ (with some predicates-related syntactic sugar) as

$$\mathsf{pull}\,i^\sharp :\equiv \mathsf{ind}_{\mathsf{Weld}}(w.D, (i \doteq 0\,?\,c.(f\ c) \mid i \doteq 1\,?\,d.d), c.(f\ c), \sqcup) : /f\backslash\, i \to D.$$

Note that modalities play an inconspicuous but utterly important role here. The predicate former $\doteq$ is continuous in its endpoints and the Weld type former is continuous in the predicate P, so that $/f\backslash$ is a bridge relating types and not a path equating them. On the other hand, weld and $\mathsf{ind}_{\mathsf{Weld}}$ are parametric in the predicate P, so that push and pull *are* heterogeneous paths, allowing us to prove parametricity theorems involving equality.

Using the Glue *operation.* Glueing is the dual operation to welding. In order to form $/f\backslash\, i$ using Glue, we need a diagram of the form $C \to E \leftarrow D$ that encodes the graph of the function f. A well-behaved relation can be represented by letting E be the disjoint union of C and D,

[4] The particular predicate $i \doteq 0 \vee i \doteq 1$ will never reduce to $\bot$ as there are no mid-interval constants, so we will never really end up in the default case. A predicate that can become false, is $i \doteq 0$.

[5] In fact, E may depend on i so we can be more general: we need functions $E[0/i] \to C$ and $E[1/i] \to D$. However, glueing and welding over constant types will be powerful enough for all applications in this paper.

quotiented by the relation. For a function, this is isomorphic to the codomain. So we set $E :\equiv D$ and $h :\equiv (i \doteq 0\,?\,f \mid i \doteq 1\,?\,\mathrm{id}_D) : T \to D$. Then we get

$$/f\backslash\, i :\equiv \mathrm{Glue}\{D \leftarrow (i \doteq 0\,?\,C, f \mid i \doteq 1\,?\,D, \mathrm{id}_D)\}, \qquad /f\backslash\, 0 \equiv C, \qquad /f\backslash\, 1 \equiv D.$$

Whereas the Weld type – like an inductive type – lets us eliminate elements by inspecting their construction, the Glue type – like a record type – lets us construct elements by saying how they eliminate. Thinking of $/f\backslash\, i$ as a system with a default clause, it is clear how we can eliminate an element b: if P holds, then b becomes an element of T; otherwise, we are in the default case and we can use a projection-like function $\mathrm{pull}\, i^\sharp :\equiv \mathrm{unglue}\,(P\,?\,h) \equiv \mathrm{unglue}\,(i \doteq 0\,?\,f \mid i \doteq 1\,?\,\mathrm{id}_D) : /f\backslash\, i \to D$. Again, this function extends to the case where P does hold, and then it specializes to h. Thus, in order to construct an element $b : /f\backslash\, i$, we need to say what element t of T it extends, and what element $a : A$ it projects to, and moreover we need that $g\, t \equiv a$. This is then assembled as $b :\equiv \mathrm{glue}\{a \leftarrowtail (P\,?\,t)\}$. In particular, we can define $\mathrm{push}\, i^\sharp :\equiv \lambda c.\mathrm{glue}\{f\, c \leftarrowtail (i \doteq 0\,?\,c \mid i \doteq 1\,?\,f\, c)\}$.

The rest of this section is dedicated to making formal every concept encountered in the above examples, to wit: the interval $\mathbb{I}$, a calculus of *face predicates* such as $i \doteq 0$, *systems* for case distinction over face predicates, and the Glue and Weld type formers. We conclude by postulating the path degeneracy axiom.

3.2.2 The Interval. The interval $\mathbb{I}$ is what we use to reason internally about bridges and paths. Although the model treats it as a type like any other, containing just two points 0 and 1 connected by a bridge, we choose to give it an exceptional syntactic treatment with the purpose of preserving the following syntactic property:

Lemma 3.3. *If $\Gamma \vdash t : \mathbb{I}$, then t is either 0, 1 or a variable from Γ with modality $\mu \leq \mathbf{id}$.* □

In other words, we want to avoid neutral interval terms. Then we should not have functions with codomain $\mathbb{I}$, and so $\mathbb{I}$ cannot be part of a universe. Instead, we consider $\Gamma \vdash i : \mathbb{I}$ a new class of judgements. So we postulate two constants $0, 1 : \mathbb{I}$ and furthermore allow: context extension with interval variables $i^\mu : \mathbb{I}$ (c-ext), use of interval variables (t-var), the construction of Π^μ- and Σ^μ-types with $\mathbb{I}$ as their domain (t-Π, t-Σ), λ-abstraction over $i^\mu : \mathbb{I}$ (t-λ) and application to interval terms (t-ap), formation of interval pairs (i^μ, y) (t-pair) and pattern matching over such pairs (t-indpair) but *not* first projections for such pairs, as they would pollute the interval term judgement.

3.2.3 Face Predicates and Face Unifiers. If we have m continuous and n parametric interval variables in the context, we can think of the term on the right as ranging over an $(m+n)$-dimensional cube that has m bridge-dimensions and n path-dimensions. A calculus of face predicates allows us to make assumptions about where we are on that cube, and to use those assumptions *definitionally*. Face predicates can be thought of as propositions, i.e. types that have at most one element. For that reason, and because we make sure face predicates are decidable, we will never explicitly write their proofs or hypotheses. Because propositions have no relational structure, we can ignore modalities for their elements. Again, while the model allows us to treat the universe of face predicates $\mathbb{F}$ as an ordinary type, we will be more restrictive in order to preserve decidability of definitional equality.

$$\frac{\Gamma \vdash i, j : \mathbb{I}}{\Gamma \vdash i \doteq j : \mathbb{F}}\text{f-eq} \qquad \frac{\Gamma \vdash \mathrm{Ctx}}{\Gamma \vdash \top : \mathbb{F}}\text{f-tt} \qquad \frac{\Gamma \vdash \mathrm{Ctx}}{\Gamma \vdash \bot : \mathbb{F}}\text{f-ff}$$

$$\frac{\Gamma \vdash P, Q : \mathbb{F}}{\Gamma \vdash P \wedge Q : \mathbb{F}}\text{f-}\wedge \qquad \frac{\Gamma \vdash P, Q : \mathbb{F}}{\Gamma \vdash P \vee Q : \mathbb{F}}\text{f-}\vee \qquad \frac{\Gamma \vdash \mathrm{Ctx} \qquad \sharp\backslash\, \Gamma \vdash P : \mathbb{F}}{\Gamma, P \vdash \mathrm{Ctx}}\text{c-f}$$

Fig. 3. Formation rules for face predicates.

Figure 3 lists the rules for generating predicates: we can equate interval terms (f-eq), we have true and false predicates (f-tt, f-ff) and conjunction and disjunction (f-$\wedge$, f-$\vee$). We can also extend contexts with a face predicate assumption (c-f). As usual, we omit congruence rules for definitional

equality. Before we can define equality for face predicates, we need two definitions. First, we say that a face predicate *holds* when its translation to a metatheoretical predicate holds, i.e. $i \doteq j$ holds if and only if $i \equiv j$, $\top$ holds, $\bot$ does not hold, $P \wedge Q$ holds when both P and Q hold, and $P \vee Q$ holds when either P or Q holds. Secondly, a *face unifier* $\sigma : \Delta \to \Gamma$ for a context Γ is a substitution of interval variables from a face-predicate-free context. To be precise, they are generated as follows:

$$\frac{}{() : () \to ()}\text{u-em} \qquad \frac{\sigma : \Delta \to \Gamma \qquad \Gamma \vdash T\,\mathsf{type}}{\sigma : (\Delta, x^\mu : T[\sigma]) \to (\Gamma, x^\mu : T)}\text{u-ext} \qquad \frac{\sigma : \Delta \to \Gamma}{\sigma : (\Delta, i^\mu : \mathbb{I}) \to \Gamma}\text{u-wkn}$$

$$\frac{\sigma : \Delta \to \Gamma \qquad \mu \setminus \Delta \vdash j : \mathbb{I}}{(\sigma, j/i) : \Delta \to (\Gamma, i^\mu : \mathbb{I})}\text{u-sub} \qquad \frac{\sigma : \Delta \to \Gamma \qquad \sharp \setminus \Gamma \vdash P : \mathbb{F} \qquad P[\sigma]\,\mathsf{holds}}{\sigma : \Delta \to (\Gamma, P)}\text{u-ext-f}$$

Given $\Gamma \vdash P, Q : \mathbb{F}$, we write $P \Rightarrow Q$ if, for every face unifier σ for Γ such that $P[\sigma]$ holds, $Q[\sigma]$ also holds. We equate predicates that are in this sense equivalent (f=). If an equality predicate is satisfied by any face unifier of the context, we can use it definitionally (i=-f).

$$\frac{\Gamma \vdash P, Q : \mathbb{F} \qquad P \Leftrightarrow Q}{\Gamma \vdash P \equiv Q : \mathbb{F}}\text{f=} \qquad \frac{\Gamma \vdash i, j : \mathbb{I} \qquad \top \Rightarrow (i \doteq j)}{\Gamma \vdash i \equiv j : \mathbb{I}}\text{i=-f}$$

The joint effect of these rules is that, as soon as there are face predicates in the context, type checking no longer happens in the current context, but only after face unification. Even though a given context has infinitely many face unifiers (by weakening), it is always sufficient to check finitely many, and further optimizations are possible. Indeed, if the face predicates in the context do not use disjunctions, then one unifier is sufficient.

Example 3.4. Terms in the context $\Gamma, i : \mathbb{I}, j : \mathbb{I}$ can be thought of as living in context Γ and varying over a two-dimensional square. If we extend the context further with the assumption $P :\equiv i \doteq 0 \vee i \doteq 1 \vee j \doteq 0 \vee j \doteq 1$, then we are restricting ourselves to the sides of that square. Terms in the extended context will be type-checked 4 times, under each of the face unifiers $(0/i), (1/i) : (\Gamma, j : \mathbb{I}) \to (\Gamma, i : \mathbb{I}, j : \mathbb{I}, P)$ and $(0/j), (1/j) : (\Gamma, i : \mathbb{I}) \to (\Gamma, i : \mathbb{I}, j : \mathbb{I}, P)$.

3.2.4 Systems. Systems are the eliminator for proofs of disjunctions ($\vee$) and contradictions ($\bot$). Assuming that $P \vee Q$ is true, we can define a term by giving its value when either P or Q is true, such that the given values match when both are true (t-sys2). Assuming a contradiction, we can spawn terms at will using the empty system (t-sys0). We also give the β-rules for systems and the η-rule for $\lightning$, which states that $\lightning$ is equal to anything.

$$\frac{\begin{array}{ll}\Gamma \vdash A\,\mathsf{type} & \Gamma, P \vdash a : A \qquad \Gamma, Q \vdash b : A \\ \Gamma, P \wedge Q \vdash a \equiv b : A & \sharp \setminus \Gamma \vdash P \vee Q \equiv \top : \mathbb{F}\end{array}}{\Gamma \vdash (P\,?\,a \mid Q\,?\,b) : A}\text{t-sys2} \qquad \frac{\begin{array}{l}\Gamma \vdash A\,\mathsf{type} \\ \sharp \setminus \Gamma \vdash \bot \equiv \top : \mathbb{F}\end{array}}{\Gamma \vdash \lightning : A}\text{t-sys0}$$

$$(\top\,?\,a \mid Q\,?\,b) \equiv a, \qquad (P\,?\,a \mid \top\,?\,b) \equiv b, \qquad \lightning \equiv a.$$

In order to avoid repeating predicates, we will denote $(P\,?\,a \mid Q \vee R\,?\,(Q\,?\,b \mid R\,?\,c))$ shorter as $(P\,?\,a \mid Q\,?\,b \mid R\,?\,c)$.

3.2.5 Welding. As clarified in Section 3.2.1, the Weld type (t-Weld) comes with a constructor (t-weld) and a variance-polymorphic eliminator (t-indweld). We supplement these rules with three equations that express that Weld and weld extend the partial objects that they should extend, as well as two β-rules for $\mathsf{ind}_{\mathsf{Weld}}$. The β-rules are compatible due to the equality required by $\mathsf{ind}_{\mathsf{Weld}}$.

$$\frac{\begin{array}{ll}\Gamma \vdash P : \mathbb{F} & \Gamma, P \vdash T : \mathcal{U}_\ell \\ \Gamma \vdash A : \mathcal{U}_\ell & \P \setminus \Gamma, P \vdash f : A \to T\end{array}}{\Gamma \vdash \mathsf{Weld}\{A \to (P\,?\,T, f)\} : \mathcal{U}_\ell}\text{t-Weld} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathsf{Weld}\{A \to (P\,?\,T, f)\}\,\mathsf{type} \\ \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathsf{weld}\,(P\,?\,f)\,a : \mathsf{Weld}\{A \to (P\,?\,T, f)\}}\text{t-weld}$$

$$\frac{\begin{array}{ll}\Gamma, y^\nu : \mathrm{Weld}\{A \to (P\,?\,T, f)\} \vdash C\,\mathsf{type} & \Gamma, P, y^\nu : T \vdash d : C \\ \Gamma, x^\nu : A \vdash c : C[\mathrm{weld}\,(P\,?\,f)\,x/y] & \Gamma, P, x^\nu : A \vdash c \equiv d[f\,x/y] : C[f\,x/y] \\ \nu \setminus \Gamma \vdash b : \mathrm{Weld}\{A \to (P\,?\,T, f)\} & \end{array}}{\Gamma \vdash \mathrm{ind}^\nu_{\mathrm{Weld}}(y.C, (P\,?\,y.d), x.c, b) : C[b/y]}\text{t-indweld}$$

$$\begin{array}{rcl@{\qquad}rcl} \mathrm{Weld}\{A \to (\top\,?\,T, f)\} & \equiv & T, & \mathrm{ind}^\nu_{\mathrm{Weld}}(y.C, (\top\,?\,y.d), x.c, b) & \equiv & d[b/y], \\ \mathrm{weld}\,(\top\,?\,f)\,a & \equiv & f\,a, & \mathrm{ind}^\nu_{\mathrm{Weld}}(y.C, (P\,?\,y.d), x.c, \mathrm{weld}\,(P\,?\,f)\,a) & \equiv & c[a/x]. \end{array}$$

3.2.6 Glueing. Dually, the Glue type (t-Glue) comes with a projection (t-unglue) and a constructor (t-glue). We supplement these with three equations stating what happens when $P \equiv \top$, a β-rule and an η-rule.

$$\frac{\begin{array}{ll}\Gamma \vdash P : \mathbb{F} & \Gamma, P \vdash T : \mathcal{U}_\ell \\ \Gamma \vdash A : \mathcal{U}_\ell & \text{¶} \setminus \Gamma, P \vdash f : T \to A\end{array}}{\Gamma \vdash \mathrm{Glue}\{A \leftarrow (P\,?\,T, f)\} : \mathcal{U}_\ell}\text{t-Glue} \qquad \frac{\Gamma \vdash b : \mathrm{Glue}\{A \leftarrow (P\,?\,T, f)\}}{\Gamma \vdash \mathrm{unglue}\,(P\,?\,f)\,b : A}\text{t-unglue}$$

$$\frac{\Gamma \vdash \mathrm{Glue}\{A \leftarrow (P\,?\,T, f)\}\,\mathsf{type} \qquad \Gamma, P \vdash t : T \qquad \Gamma \vdash a : A \qquad \Gamma, P \vdash f\,t \equiv a : A}{\Gamma \vdash \mathrm{glue}\{a \leftarrowtail (P\,?\,t)\} : \mathrm{Glue}\{A \leftarrow (P\,?\,T, f)\}}\text{t-glue}$$

$$\begin{array}{rcl@{\qquad}rcl} \mathrm{Glue}\{A \leftarrow (\top\,?\,T, f)\} & \equiv & T, & \mathrm{unglue}\,(P\,?\,f)\,(\mathrm{glue}\{a \leftarrowtail (P\,?\,t)\}) & \equiv & a, \\ \mathrm{glue}\{a \leftarrowtail (\top\,?\,t)\} & \equiv & t, & \mathrm{glue}\{(\mathrm{unglue}\,(P\,?\,f)\,b) \leftarrowtail (P\,?\,b)\} & \equiv & b. \\ \mathrm{unglue}\,(\top\,?\,f)\,b & \equiv & f\,b, & & & \end{array}$$

3.2.7 From Identity Extension to the Path Degeneracy Axiom. Relational parametricity in System F [Reynolds 1983] asserts that, at both the type and the value level, related inputs will lead to related outputs; thus, 'relatedness' is a congruence. In this formulation, to say that types A_1 and A_2 are related, is to give a relation $[A] : \mathrm{Rel}(A_1, A_2)$. Relatedness of values $a_1 : A_1$ and $a_2 : A_2$ is then defined by that same relation $[A]$. The identity extension lemma (IEL) asserts that if, at the type level, all inputs are of the form $\mathrm{Eq}_A : \mathrm{Rel}(A, A)$, then the output will also be of that form. In particular, relatedness in a closed type (with no type-level input) means equality. Then 'relatedness' at the value level can be thought of as heterogeneous equality: it is a congruence that boils down to equality when it becomes homogeneous. However, 'relatedness' at the type level does not mean equality, as $\mathrm{Rel}(A, B)$ can be inhabited for any two types A and B. This explains the difficulty in extending IEL to large types in dependent type theory.

Our approach in the transition to dependent types, is to mix value and type levels, while maintaining two separate relations: the bridge relation is like type-level relatedness in System F, whereas the path relation is like value-level relatedness in System F and expresses heterogeneous equality. Parametric functions in System F map types to values; hence our parametric functions map bridges to paths. We already know that all functions preserve paths (since $\mu \circ \sharp = \sharp$), so we only need to add that path-connectedness in closed types means equality. We assert this by postulating that all non-dependent paths are constant, implying that their endpoints are equal:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \sharp \setminus \Gamma \vdash p : \forall(i : \mathbb{I}).A}{\Gamma \vdash \mathrm{degax}\,p : p =_{\forall(i:\mathbb{I}).A} \left(\lambda(i^\sharp : \mathbb{I}).p\,0^\sharp\right)}\text{t-degax}$$

3.3 Related and Unrelated Naturals

We include two types Nat and Size which both represent the natural numbers, but with different relational structure. In Nat, numbers are only related to themselves, i.e. every bridge $\mathbb{I} \to \mathrm{Nat}$ is constant. In Size, any two sizes are related, i.e. the bridge relation is codiscrete. As such, it is easier to create functions of domain Nat, but these functions come with fewer type-guaranteed properties. The type of naturals Nat is highly similar to that of MLTT:

$$\frac{\Gamma \vdash \mathsf{Ctx}}{\Gamma \vdash \mathsf{Nat} : \mathcal{U}_0}\text{t-Nat} \qquad \frac{\Gamma, m^\nu : \mathsf{Nat} \vdash C\,\mathsf{type} \quad \Gamma \vdash c_0 : C[0/m] \quad \Gamma, m^\nu : \mathsf{Nat}, c : C \vdash c_\mathsf{s} : C[\mathsf{s}\,m/m] \quad \nu \setminus \Gamma \vdash n : \mathsf{Nat}}{\Gamma \vdash \mathsf{ind}^\nu_{\mathsf{Nat}}(m.C, c_0, m.c.c_\mathsf{s}, n) : C[n/m]}\text{t-indnat}$$

$$\frac{\Gamma \vdash \mathsf{Ctx}}{\Gamma \vdash 0 : \mathsf{Nat}}\text{t-0} \qquad \frac{\Gamma \vdash n : \mathsf{Nat}}{\Gamma \vdash \mathsf{s}\,n : \mathsf{Nat}}\text{t-s}$$

We omit the β-rules. Perhaps surprisingly, the eliminator allows us to create parametric functions from the naturals by pattern matching. For example, we can have a parametric identity function:

$$\lambda n^\sharp.\mathsf{ind}^\sharp_{\mathsf{Nat}}(m.\mathsf{Nat}, 0, m.c.(\mathsf{s}\,c), n) : \sharp\mathsf{Nat} \to \mathsf{Nat}. \tag{2}$$

This makes sense, because Nat has no interesting bridges, so every function trivially maps bridges to paths, as required by the parametric modality $\sharp$. This is in line with the known theorem [Atkey et al. 2014; Takeuti 2001] that any function in MLTT with small codomain, is parametric, even though such functions may pattern match on natural numbers. The reason for the surprise may lie in the fact that parametricity over a discrete domain typically does not arise in e.g. System Fω, where elements of the most common kinds are always related. In Section 6, we contrast parametricity with irrelevance.

The type Size, in contrast, has the following constructors:

$$\frac{\Gamma \vdash \mathsf{Ctx}}{\Gamma \vdash \mathsf{Size} : \mathcal{U}_0}\text{t-Size} \qquad \frac{\Gamma \vdash \mathsf{Ctx}}{\Gamma \vdash 0_\mathsf{S} : \mathsf{Size}}\text{t-Size-0} \qquad \frac{\Gamma \vdash n : \mathsf{Size}}{\Gamma \vdash \uparrow n : \mathsf{Size}}\text{t-Size-s} \qquad \frac{\Gamma \vdash P : \mathbb{F} \quad \Gamma, P \vdash n : \mathsf{Size}}{\Gamma \vdash \mathsf{fill}(P\,?\,n) : \mathsf{Size}}\text{t-Size-fill}$$

where $\mathsf{fill}(P\,?\,n)$ extends n, i.e. $\mathsf{fill}(\top\,?\,n) \equiv n : \mathsf{Size}$. Moreover, we have $\mathsf{fill}(\bot\,?\,n) \equiv 0_\mathsf{S}$. So the fill operation completes a partial size by adding 0_S in missing vertices, and filling up the relational parts using the fact that any two sizes are related. However, we want more: if the bridge relation is to be codiscrete, then the relational action of any continuous function to Size should be void of information. This means not only that any two sizes are related, as is asserted by $\lambda i.\mathsf{fill}(i \doteq 0\,?\,m \mid i \doteq 1\,?\,n)$, but also that they are related in a unique way. To that end, we add the following, somewhat unconventional typing rule:

$$\frac{(i^\mu : \mathbb{I}) \in \Gamma \qquad \mu \leq \mathbf{id} \qquad \Gamma \vdash m, n : \mathsf{Size} \qquad \Gamma, (i \doteq 0 \vee i \doteq 1) \vdash m \equiv n : \mathsf{Size}}{\Gamma \vdash m \equiv n : \mathsf{Size}}\text{t=-Size-codisc}$$

Repeated application shows that two sizes are equal as soon as they are equal on all bridge-vertices of Γ, i.e. they become equal after substituting every continuous and pointwise interval variable with a constant.

An eliminator that simply takes images for each of these constructors, satisfying the necessary equations, would be very complicated to use. Instead, we choose to provide the strong principle of induction (t-fix) which uses an inequality type (t-$\leq$). Since instances of the fix^ν combinator expand non-terminatingly, we include its β-rule as a (non-computational) equality axiom (t-fix-eq), although the model supports the definitional equality.

$$\frac{\Gamma, n^\nu : \mathsf{Size} \vdash A\,\mathsf{type} \qquad \Gamma \vdash f : \Pi^\nu(n : \mathsf{Size}).(\Pi^\nu(m : \mathsf{Size}).(\uparrow m \leq n) \to A[m/n]) \to A}{\Gamma \vdash \mathsf{fix}^\nu\, f : \Pi^\nu(n : \mathsf{Size}).A}\text{t-fix}$$

$$\frac{\sharp \setminus \Gamma \vdash \mathsf{fix}^\nu\, f : \Pi^\nu(n : \mathsf{Size}).A \qquad (\sharp \circ \nu) \setminus \Gamma \vdash m : \mathsf{Size}}{\Gamma \vdash \mathsf{fix}^\nu_=\, f\, m : \mathsf{fix}^\nu\, f\, m^\nu =_A f\, m^\nu\, (\lambda n^\nu.\lambda e.\mathsf{fix}^\nu\, f\, n^\nu)}\text{t-fix-eq} \qquad \frac{\Gamma \vdash m, n : \mathsf{Size}}{\Gamma \vdash m \leq n : \mathcal{U}_0}\text{t-}\leq$$

$$\frac{\sharp \setminus \Gamma \vdash n : \mathsf{Size}}{\Gamma \vdash \mathsf{refl}_\leq\, n : n \leq n}\text{t-}\leq\text{-refl} \qquad \frac{\Gamma \vdash e_1 : n_0 \leq n_1 \qquad \Gamma \vdash e_2 : n_1 \leq n_2}{\Gamma \vdash \mathsf{trans}_\leq\, e_1\, e_2 : n_0 \leq n_2}\text{t-}\leq\text{-trans}$$

$$\frac{\sharp \setminus \Gamma \vdash n : \mathsf{Size}}{\Gamma \vdash \mathsf{zero}_\leq\, n : 0 \leq n}\text{t-}\leq\text{-zero} \qquad \frac{\Gamma \vdash e : m \leq n}{\Gamma \vdash \mathsf{step}_\leq\, e : \uparrow m \leq \uparrow n}\text{t-}\leq\text{-step}$$

$$\frac{\Gamma, P \vdash e : m \leq n}{\Gamma \vdash \mathsf{fill}_{\leq}(P\,?\,e) : \mathsf{fill}(P\,?\,m) \leq \mathsf{fill}(P\,?\,n)}\text{t-}\leq\text{-fill} \qquad \begin{array}{lcl} \mathsf{fill}_{\leq}(\top\,?\,e) & \equiv & e \\ \mathsf{fill}_{\leq}(\bot\,?\,e) & \equiv & \mathsf{zero}_{\leq}\,0_{\mathsf{S}} \end{array}$$

$$\frac{(i^{\mu} : \mathbb{I}) \in \Gamma \qquad \Gamma \vdash e, e' : m \leq n \qquad \Gamma, (i \doteq 0 \vee i \doteq 1) \vdash e \equiv e' : m \leq n}{\Gamma \vdash e \equiv e' : m \leq n}\text{t=-}\leq\text{-codisc}$$

The combinator fix^{ν} allows us to define $g\,n^{\nu}$ in terms of the restriction of g to $\mathsf{Size}_{<n}$:

$$g|_{<n} :\equiv \lambda(m^{\nu} : \mathsf{Size}).\lambda(e : \uparrow m \leq n).g\,m^{\nu}. \tag{3}$$

An intuitive argument why fix^{ν} can have the modalities it does, is the following: since we have $\mathsf{fix}^{\nu}\,f\,n^{\nu} = f\,n^{\nu}\,(\lambda m^{\nu}.\lambda e.\mathsf{fix}^{\nu}\,f\,m^{\nu})$ and the right hand side is ν-modal in n, so is the left hand side. This reasoning shows that, when we pass a bridge in the argument n, the action of ν on bridges will be respected during expansion of fix^{ν}. Note that although the 0_{S} and $\uparrow$ constructors allow us to create a function $\mathsf{Nat} \to \mathsf{Size}$, we cannot construct an inverse. Indeed, fix^{ν} does not allow arbitrary distinctions between 0_{S} and $\uparrow n$.

We could add many more Size-related primitives, e.g. for deriving contradictions from assumptions like $e : \uparrow n \leq n$ or to prove that all inequality proofs are definitionally equal. However, the current set of rules, combined with the following maximum operator (t-$\sqcup$) is more than sufficient for the practical applications in Section 4.2.

$$\frac{\Gamma \vdash m, n : \mathsf{Size}}{\Gamma \vdash m \sqcup n : \mathsf{Size}}\text{t-}\sqcup \qquad \frac{\sharp \leq \Gamma \vdash m, n : \mathsf{Size}}{\Gamma \vdash \mathsf{lmax}_{\leq}\,m\,n : m \leq m \sqcup n}\text{t-}\leq\text{-}\sqcup\text{-l} \qquad \frac{\sharp \leq \Gamma \vdash m, n : \mathsf{Size}}{\Gamma \vdash \mathsf{rmax}_{\leq}\,m\,n : n \leq m \sqcup n}\text{t-}\leq\text{-}\sqcup\text{-r}$$

3.4 Embedding Other Systems

To understand precisely how powerful our type system is, it is useful to compare it to others. In this section, we show that MLTT and a predicative variant of System Fω can be embedded into our system using either the continuous or the pointwise modality. The latter shows that the pointwise modality of Glue and Weld's dependency on the diagram arrows will only interfere with special features of ParamDTT and not with features already present in MLTT or System Fω. Indeed, everything that can be done in MLTT, can be done pointwise in ParamDTT. Each of these results can be proven by induction on the derivation tree of the translated judgement.

Lemma 3.5 (Embedding of MLTT). *Let μ be either* **id** *or* ¶. *Then every derivable judgement of MLTT can be translated to a derivable judgement of ParamDTT by inserting μ wherever a modality is required.*[6] *Thus, we have extended MLTT.*

We can make System Fω [see e.g. Pierce 2002, ch. 30] predicative by annotating the kind $*$ with a level $\ell \in \mathbb{N}$ and assigning levels to types in the style of MLTT, e.g. if $\varphi : *_j \to *_k \vdash A : *_{\ell}$, then $\forall(\varphi : *_j \to *_k).A : *_{\max\{j+1,k+1,\ell\}}$. This extends the predicative variant of System F by Leivant [1991].

Lemma 3.6 (Embedding of predicative System Fω). *Let μ be either* **id** *or* ¶. *Then every derivable judgement of predicative System Fω can be translated to a derivable judgement of ParamDTT by the tables in Fig. 4 (where we omit the translation of terms and equality judgements).*[7]

Kinds	$\to$	Types
$[\![*_{\ell}]\!]$	=	$\mathcal{U}_{\ell}$
$[\![\kappa \to \kappa']\!]$	=	$[\![\kappa]\!] \to [\![\kappa']\!]$

Kinding contexts	$\to$	Contexts
$[\![()]\!]$	=	$()$
$[\![\Delta, \alpha : \kappa]\!]$	=	$[\![\Delta]\!], \alpha^{\sharp} : [\![\kappa]\!]$

Contexts	$\to$	Contexts
$[\![\Delta \mid ()]\!]$	=	$[\![\Delta]\!]$
$[\![\Delta \mid \Gamma, x : A]\!]$	=	$[\![\Delta \mid \Gamma]\!], x^{\mu} : A$

Types	$\to$	Types
$[\![\forall(\alpha : \kappa).A]\!]$	=	$\forall(\alpha : [\![\kappa]\!]).[\![A]\!]$
$[\![\exists(\alpha : \kappa).A]\!]$	=	$\exists(\alpha : [\![\kappa]\!]).[\![A]\!]$
$[\![A \to B]\!]$	=	$\mu[\![A]\!] \to [\![B]\!]$
$[\![A \times B]\!]$	=	$\mu[\![A]\!] \times [\![B]\!]$

Judgements	$\to$	Judgements
$[\![\Delta \vdash \mathsf{KindingCtx}]\!]$	=	$[\![\Delta]\!] \vdash \mathsf{Ctx}$
$[\![\Delta \mid \Gamma \vdash \mathsf{Ctx}]\!]$	=	$[\![\Delta \mid \Gamma]\!] \vdash \mathsf{Ctx}$
$[\![\Delta \vdash T : \kappa]\!]$	=	$\sharp \setminus [\![\Delta]\!] \vdash [\![T]\!] : [\![\kappa]\!]$
$[\![\Delta \mid \Gamma \vdash t : T]\!]$	=	$[\![\Delta \mid \Gamma]\!] \vdash [\![t]\!] : [\![T]\!]$

Fig. 4. Embedding predicative System Fω into ParamDTT.

4 APPLICATIONS

Mechanized Agda proofs for the results of this section, are available in the artifact or online[8].

[6] Alternatively, all Σ-types can be annotated with **id** instead of μ.
[7] Alternatively, all product types can be annotated with **id** instead of μ.
[8] https://github.com/Saizan/parametric-demo

4.1 Church Encoded (Co)-Recursive Types

In System F, we can use Church encoding to represent data types. For example, for a fixed type B, we can encode the type of lists over B as $\mathsf{ChList}\ B :\equiv \forall\alpha.\alpha \to (B \to \alpha \to \alpha) \to \alpha$. The list $[b_1, b_2]$ is then encoded as $\Lambda\alpha.\lambda\mathsf{nil}'.\lambda\mathsf{cons}'.\mathsf{cons}'\ b_1\ (\mathsf{cons}'\ b_2\ \mathsf{nil}')$. With dependent types, we would define $\mathsf{ChList}_\ell\ B :\equiv \Pi(X : \mathcal{U}_\ell).X \to (B \to X \to X) \to X$. Now if B happens to be some universe $\mathcal{U}_j$ with $j \geq \ell$, then the following term should arouse suspicion:

$$\mathsf{exoticList} = \lambda X.\lambda\mathsf{nil}'.\lambda\mathsf{cons}'.\mathsf{cons}'\ X\ \mathsf{nil}' : \mathsf{ChList}_\ell\ \mathcal{U}_j$$

If this is a list of types, then it is one whose contents depend on the type X which we are eliminating to. This may not be detrimental – for example an analysis based on universe levels may reveal that we cannot extract the contents of such a list if $j \geq \ell$. However, using $\forall$ instead of Π, we can instead forbid exoticList altogether.

In this section, we use parametric Church encoding to construct indexed (co)-recursive data types. More formally put, we construct, up to universe level issues, initial algebras and final co-algebras for indexed functors, and prove their universal properties internally. We assume throughout this section that we have a context Γ and an index type $\sharp \setminus \Gamma \vdash Z : \mathcal{U}_0$ (which we take, for simplicity, in $\mathcal{U}_0$ although our formal proof in Agda is more general). We will write $X \Rightarrow Y$ for $\forall(z : Z).X\ z \to Y\ z$ and $\circledcirc$ for the corresponding composition operator.[9] We moreover assume a functor F which is a scheme consisting of

$$\sharp \setminus \Gamma \vdash F_\ell : (Z \to \mathcal{U}_\ell) \to (Z \to \mathcal{U}_\ell)$$
$$¶ \setminus \Gamma \vdash \vec{F}^\ell_k : \forall(X : Z \to \mathcal{U}_k).\forall(Y : Z \to \mathcal{U}_\ell).(X \Rightarrow Y) \to (F\,X \Rightarrow F\,Y)$$

for all metatheoretic $k, \ell \in \mathbb{N}$, such that identity and composition are preserved definitionally, even across universes. We will omit level annotations on F and $\vec{F}$, as well as the first two arguments of $\vec{F}$. Throughout the section, we will write $t^\mu : T$ for $\mu \setminus \Gamma \vdash t : T$. Note that theorems are strongest when their assumptions are parametric and their conclusions are pointwise.

4.1.1 *Initial Algebras.*[10] Define

$$\sharp \setminus \Gamma \vdash \mathsf{Mu}_\ell :\equiv \lambda z.\forall(X : Z \to \mathcal{U}_\ell).¶(F\,X \Rightarrow X) \to X\ z : Z \to \mathcal{U}_{\ell+1},$$
$$\Gamma \vdash \mathsf{fold} :\equiv \lambda X^\sharp.\lambda\mathsf{mk}X^¶.\lambda z^\sharp.\lambda m.m\ X^\sharp\ \mathsf{mk}X^¶ : \forall(X : Z \to \mathcal{U}_\ell).¶(F\,X \Rightarrow X) \to (\mathsf{Mu}_\ell \Rightarrow X),$$
$$\Gamma \vdash \mathsf{mkMu}_\ell :\equiv \lambda z^\sharp.\lambda\widehat{m}.\lambda X^\sharp.\lambda\mathsf{mk}X^¶.(\mathsf{mk}X \circledcirc \vec{F}\,(\mathsf{fold}\,X^\sharp\ \mathsf{mk}X^¶))\ z^\sharp\ \widehat{m} : F\,\mathsf{Mu}_\ell \Rightarrow \mathsf{Mu}_\ell,$$
$$\Gamma \vdash \downarrow :\equiv \lambda z^\sharp.\lambda m.\lambda X^\sharp.\lambda\mathsf{mk}X^¶.m\ X^\sharp\ \mathsf{mk}X^¶ : \mathsf{Mu}_{\ell+1} \Rightarrow \mathsf{Mu}_\ell.$$

Lemma 4.1 (Naturality of fold). *Assume $A^\sharp, B^\sharp : Z \to \mathcal{U}_\ell$ with algebra structures $\mathsf{mk}A^¶ : F\,A \Rightarrow A$ and $\mathsf{mk}B^¶ : F\,B \Rightarrow B$. Assume a morphism of algebras $f^¶ : A \Rightarrow B$ (i.e. $f \circledcirc \mathsf{mk}A \equiv \mathsf{mk}B \circledcirc \vec{F}\,f$). Then there exists a term $\mathsf{natlemma}^¶ : f \circledcirc \mathsf{fold}\,A^\sharp\ \mathsf{mk}A^¶ =_{\mathsf{Mu}_\ell \Rightarrow B} \mathsf{fold}\,B^\sharp\ \mathsf{mk}B^¶$.*

Sketch of proof. The proof boils down to the construction of the diagram of algebra morphisms in Fig. 5, where $/f\setminus$ differs from the type defined in Section 3.2.1 using Glue in that it is now a Z-indexed type formed from a Z-indexed function, i.e. we currently write $/f\setminus\ i\ z$ instead of $/f\ z^\sharp\setminus\ i$. The path degeneracy axiom then asserts that the top and bottom rows compose to equal functions.

A major hurdle is that we need a proof $(\mathsf{mk}/f\setminus\ i^\sharp)^¶ : F(/f\setminus\ i) \Rightarrow /f\setminus\ i$ that $/f\setminus\ i$ is an algebra, so that we can apply fold. Here, the push and pull functions are insufficient, and it becomes important that we use the Glue-implementation of $/f\setminus$. Indeed, the weld constructor only takes input of type

[9] One can argue that it is more general to consider continuously indexed functions: $\Pi(z : Z).X\ z \to Y\ z$. However, those interfere with the ¶ modality. In Section 4.2, we do use continuously indexed functions.

[10] For initial algebras, we have been able to refine the proofs so as to obtain the same results, but with continuous instead of pointwise dependencies. There are some complications in dualizing the technique we used for final co-algebras.

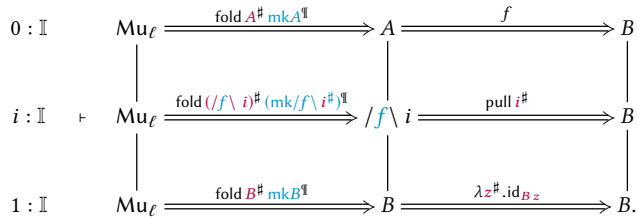

Fig. 5. Proving naturality of fold. The middle row depends on i and reduces to the top/bottom row for $i \doteq 0$ and $i \doteq 1$ respectively.

A, and applying $\mathsf{ind}_{\mathsf{Weld}}$ under the functor F would not allow us to escape F. However, we do have $\mathsf{mk}B \circledcirc \vec{F}\ \mathsf{pull}\ i^\sharp{}^\P : F(/f\setminus i) \Rightarrow B$, which is enough for glue. See the Agda proofs for details. □

We can prove, up to universe level issues, that folding to Mu is the identity:

Lemma 4.2. *We have* $\mathsf{loweringlemma}^\P : \downarrow =_{\mathsf{Mu}_{\ell+1} \Rightarrow \mathsf{Mu}_\ell} \mathsf{fold}_{\ell+1}\ \mathsf{Mu}_\ell{}^\sharp\ \mathsf{mkMu}_\ell{}^\P$.

Proof. By function extensionality, it is sufficient to prove the equation only when postcomposed with a general $\mathsf{fold}_\ell\ X^\sharp\ \mathsf{mk}X^\P : \mathsf{Mu}_\ell \Rightarrow X$. It is clear that $\mathsf{fold}_\ell\ X^\sharp\ \mathsf{mk}X^\P \circledcirc \downarrow$ is equal to $\mathsf{fold}_{\ell+1}\ X^\sharp\ \mathsf{mk}X^\P$, so we can apply the previous theorem to $\mathsf{fold}_\ell\ X^\sharp\ \mathsf{mk}X^\P$. □

Combining these lemmas, we can conclude up to universe level issues that $\mathsf{fold}\ B^\sharp\ \mathsf{mk}B^\P$ is the only algebra morphism $\mathsf{Mu} \Rightarrow B$:

Theorem 4.3. *Assume we have* $B^\sharp : Z \to \mathcal{U}_{\ell+1}$, *an algebra structure* $\mathsf{mk}B^\P : F\,B \Rightarrow B$ *and an algebra morphism* $f^\P : \mathsf{Mu}_\ell \Rightarrow B$ *(i.e.* $f \circledcirc \mathsf{mkMu}_\ell \equiv \mathsf{mk}B \circledcirc \vec{F} f$*). Then* $\mathsf{initiality}^\P : f \circledcirc \downarrow =_{\mathsf{Mu}_{\ell+1} \Rightarrow B} \mathsf{fold}_{\ell+1}\ B^\sharp\ \mathsf{mk}B^\P$. □

This theorem is perhaps less interesting than the proof technique. For example, the type of Church lists $\forall(X : \mathcal{U}_\ell).\P X \to \P(B \to X \to X) \to X$ and the Church-encoded unit type $\forall(X : \mathcal{U}_\ell).\P X \to X$ mentioned in Section 2 do not precisely fit the constraints of the theorem. However, they do up to isomorphism if there is a unit type and a coproduct type former, and the proof technique even applies if those types do not exist. What we have demonstrated is in fact a more general thesis, namely that Church encodings of indexed recursive types can be shown internally to work.

4.1.2 Final Co-algebras. Define, using some syntactic sugar for (elimination of) triples:

$\sharp \setminus \Gamma \vdash \mathsf{Nu}_\ell :\equiv \lambda(z : Z).\exists(X : Z \to \mathcal{U}_\ell).\P(X \Rightarrow F\,X) \times X\ z : Z \to \mathcal{U}_{\ell+1}$,

$\Gamma \vdash \mathsf{unfold} :\equiv \lambda X^\sharp.\lambda \mathsf{match}X^\P.\lambda z^\sharp.\lambda x.(X^\sharp, \mathsf{match}X^\P, x) : \forall(X : \mathcal{U}_\ell).\P(X \Rightarrow F\,X) \to (X \Rightarrow \mathsf{Nu}_\ell)$,

$\Gamma \vdash \mathsf{matchNu}_\ell :\equiv \lambda z^\sharp.\lambda(X^\sharp, \mathsf{match}X^\P, x).(\vec{F}\,(\mathsf{unfold}\ X^\sharp\ \mathsf{match}X^\P) \circledcirc \mathsf{match}X)\ z^\sharp\ x : \mathsf{Nu}_\ell \Rightarrow F\,\mathsf{Nu}_\ell$,

$\Gamma \vdash \uparrow :\equiv \lambda z^\sharp.\lambda(X^\sharp, \mathsf{match}X^\P, x).(X^\sharp, \mathsf{match}X^\P, x) : \mathsf{Nu}_\ell \Rightarrow \mathsf{Nu}_{\ell+1}$.

Then similar reasoning as before, but with $/\sqcup\setminus$ implemented using Weld instead of Glue, shows:

Theorem 4.4. *Assume we have* $B^\sharp : Z \to \mathcal{U}_{\ell+1}$, *a co-algebra structure* $\mathsf{match}B^\P : B \Rightarrow F\,B$ *and a co-algebra morphism* $f^\P : B \Rightarrow \mathsf{Nu}_\ell$ *(i.e.* $\mathsf{matchNu}_\ell \circledcirc f \equiv \vec{F} f \circledcirc \mathsf{match}B$*). Then* $\mathsf{finality}^\P : \uparrow \circledcirc f =_{B \Rightarrow \mathsf{Nu}_{\ell+1}} \mathsf{unfold}\ B^\sharp\ \mathsf{match}B^\P$.

4.2 Sized Types

As an example of the use of parametric quantification over values, in this section we show how it can express irrelevance properties of definitions that use sized types. By indexing data-types with a bound to the height of their elements, sized types reduce both terminating recursion and productive co-recursion to well-founded induction on sizes [Abel and Pientka 2013]. This allows to enforce

	universal			existential		
a	$\forall \_.T$	$\cong$	T	$\exists \_.T$	$\cong$	T
b	$\forall n.A\,n$	$\cong$	$\forall n.\forall m < n.A\,m$	$\exists n.A\,n$	$\cong$	$\exists n.\exists m < n.A\,m$
c	$\Pi(x : T).\forall n.B\,x\,n$	$\cong$	$\forall n.\Pi(x : T).B\,x\,n$	$\Sigma(x : T).\exists n.B\,x\,n$	$\cong$	$\exists n.\Sigma(x : T).B\,x\,n$
d	$(\forall n.A\,n) \times (\forall n.B\,n)$	$\cong$	$\forall n.(A\,n \times B\,n)$	$(\exists n.A\,n) + (\exists n.B\,n)$	$\cong$	$\exists n.(A\,n + B\,n)$
e	$\forall n.\Sigma(x : T).B\,x\,n$	$\cong$	$\Sigma(x : T).\forall n.B\,x\,n$			
f	$\forall n.(A\,n + B\,n)$	$\cong$	$(\forall n.A\,n) + (\forall n.B\,n)$			
g				$(\exists n.A\,n) \times (\exists n.B\,n)$	$\cong$	$\exists n.(A^< n \times B^< n)$
				where $C^< n$	$:\equiv$	$\exists m < n.C\,m$

Fig. 6. Isomorphisms for working with Size quantifiers. Isomorphisms between dually quantified types are placed side by side.

the totality of programs through typing and allows more recursion patterns to be recognized as well-founded when compared to more syntactic checkers.

However, while these sizes are just natural (or in some applications ordinal) numbers, they require a different treatment than other natural numbers, as a bound on the length of a list should not be considered computational content of that list. Consider the following sized list type and its constructors:

$$\begin{array}{lcl} \mathsf{SList}\,A & : & \mathsf{Size} \to \mathcal{U} \\ \mathsf{nil} & : & \Pi(n : \mathsf{Size}).\Pi(m : \mathsf{Size}).(m < n) \to \mathsf{SList}\,A\,n \\ \mathsf{cons} & : & \Pi(n : \mathsf{Size}).\Pi(m : \mathsf{Size}).(m < n) \to A \to \mathsf{SList}\,A\,m \to \mathsf{SList}\,A\,n \end{array}$$

If we treat sizes as computational content, then we obtain two different empty lists of size 2:

$$\mathsf{n}_0 :\equiv \mathsf{nil}\,(\uparrow\uparrow 0_\mathsf{S})\,0_\mathsf{S}\,(\ldots), \qquad \mathsf{n}_1 :\equiv \mathsf{nil}\,(\uparrow\uparrow 0_\mathsf{S})\,(\uparrow 0_\mathsf{S})\,(\ldots).$$

In ParamDTT we can fix this problem by making the constructors parametric in their size arguments. Then we can use the codiscrete bridge structure of Size, to build a path $\lambda i^\sharp.\mathsf{nil}\,(\uparrow\uparrow 0_\mathsf{S})^\sharp\,(\mathsf{fill}(i \doteq 0\,?\,0_\mathsf{S} \mid i \doteq 1\,?\,\uparrow 0_\mathsf{S}))^\sharp\,(\ldots)$ from n_0 to n_1. To show that this approach to sized types is valid we will build initial algebras and final co-algebras, and their sized version, for sufficiently well-behaved functors. Our approach is immature for practical use as it relies on many propositional equalities, some of which are non-computing axioms. Throughout the section, we will omit the domain of quantifiers over Size, and we will moreover write $\exists m < n.T$ for $\exists m.(m < n) \times T\,m$ and $\forall m < n.T$ for $\forall m.(m < n) \to T$.

4.2.1 Isomorphisms for Size Quantifiers. To do so we will make use of the isomorphisms in Fig. 6 which describe how parametric quantification over Size commutes with other connectives. We highlight the central ideas here, see the full Agda proofs for details. We get to remove quantifiers over constant types (a), because inserting different values of n into function applications $f\,n^\sharp$ or existential pairs $(n^\sharp, t)$ yields path-connected and hence equal values. The reasoning for (b) is similar. The isomorphisms (c) involve trivial swapping of arguments or components. In the special case where $T = \mathsf{Bool}$, we get (d) modulo a trivial isomorphism. In the isomorphism (e), the first component is constant by (a) and can be extracted from the quantifier. Specializing to Bool again, we arrive at (f). To the right of (e), we would like to write that $\exists n.\Pi(x : T).B\,x\,n$ were isomorphic to $\Pi(x : T).\exists n.B\,x\,n$. However, this would require an operator that joins T many sizes into a single size $n_\sqcup$, and moreover a way to transport from various types $B\,x\,n$ to the type $B\,x\,n_\sqcup$. This is possible when $T = \mathsf{Bool}$ by using the maximum operator ($\sqcup$), provided that $B\,x\,n$ is covariant in n. This covariance is clear if $B\,x$ is of the form $C^<$, in which case we can use (b) to extract C on the left. This yields (g).

Example 4.5. We build a fixpoint for the functor $\top + (A \times \sqcup)$, assuming that we already have the sized initial algebra $\mathsf{SList}\,A : \mathsf{Size} \to \mathcal{U}$ such that $\mathsf{SList}\,A\,n \cong \top + A \times (\exists m < n.\mathsf{SList}\,A\,m)$. We have

$$\exists n.\mathsf{SList}\,A\,n \cong \exists n.(\top + A \times (\exists m < n.\mathsf{SList}\,A\,m)) \cong_{\mathrm{d}} (\exists n.\top) + \exists n.(A \times (\exists m < n.\mathsf{SList}\,A\,m))$$
$$\cong_{\mathrm{a,c}} \top + A \times \exists n.\exists m < n.\mathsf{SList}\,A\,m \cong_{\mathrm{b}} \top + A \times \exists n.\mathsf{SList}\,A\,n$$

4.2.2 Sized Initial Algebras. In this section we fix a universe level ℓ, a context Γ and a type $\sharp \setminus \Gamma \vdash Z : \mathcal{U}_\ell$ and take an Z-indexed functor to be a pair of terms

$$\sharp \setminus \Gamma \vdash F : (Z \to \mathcal{U}_\ell) \to (Z \to \mathcal{U}_\ell)$$
$$\Gamma \vdash \vec{F} : \forall(A, B : Z \to \mathcal{U}_\ell).\,(\Pi(z : Z).\,A\,z \to B\,z) \to \Pi(z : Z).\,F\,A\,z \to F\,B\,z$$

such that $\vec{F}$ satisfies the identity and composition laws propositionally. We omit parametric type arguments such as the first two arguments to $\vec{F}$. Write $(A \Rightarrow B)$ for $(\forall n.\Pi(z : Z).\,A\,n\,z \to B\,n\,z)$. We lift the functor F to a functor $\hat{F}$ on the category of sized indexed types as follows (using some syntactic sugar for eliminating pairs):

$$\sharp \setminus \Gamma \vdash \hat{F} :\equiv \lambda A.\lambda n.F(\lambda z.\exists m < n.A\,m\,z) : (\mathsf{Size} \to Z \to \mathcal{U}_\ell) \to (\mathsf{Size} \to Z \to \mathcal{U}_\ell)$$
$$\Gamma \vdash \vec{\hat{F}} :\equiv \lambda A^\sharp.\lambda B^\sharp.\lambda f.\lambda n^\sharp.\vec{F}(\lambda z.\lambda(m^\sharp, e, a).(m^\sharp, e, f\,m^\sharp\,z\,a)) : \forall A, B.\,(A \Rightarrow B) \to \hat{F}\,A \Rightarrow \hat{F}\,B$$

which can be shown to also respect the functor laws. We then define the sized initial algebra $\widehat{\mathsf{Mu}}$ as the unique fixpoint of $\hat{F}$, using well-founded induction on Size:

$$\sharp \setminus \Gamma \vdash \widehat{\mathsf{Mu}} :\equiv \mathsf{fix}\,(\lambda n.\lambda \mathsf{Mu}'.F(\lambda z.\exists(m : \mathsf{Size}).\Sigma(e : m < n).\mathsf{Mu}'\,m\,e\,z)) : \mathsf{Size} \to Z \to \mathcal{U}_\ell.$$

From $\mathsf{fix}_=$ we obtain the propositional equality $\widehat{\mathsf{Mu}} = \hat{F}\,\widehat{\mathsf{Mu}}$ which gives us the (invertible) algebra structure $\Gamma \vdash \mathsf{mk}\widehat{\mathsf{Mu}} : \hat{F}\,\widehat{\mathsf{Mu}} \Rightarrow \widehat{\mathsf{Mu}}$, while the initiality is given by the following $\widehat{\mathsf{fold}}$ function:

$$\Gamma \vdash \widehat{\mathsf{fold}} : \forall A.(\hat{F}\,A \Rightarrow A) \to (\widehat{\mathsf{Mu}} \Rightarrow A), \qquad \widehat{\mathsf{fold}}\,A^\sharp\,\mathsf{mk}A :\equiv$$
$$\mathsf{fix}^\sharp\Big(\lambda n^\sharp.\lambda \mathsf{fold}'.\lambda z.\lambda mu.\mathsf{mk}A\,n^\sharp\,z\,\Big(\vec{F}\,(\lambda z'.\lambda(m^\sharp, e, mu').(m^\sharp, e, \mathsf{fold}'\,m^\sharp\,e\,z'\,mu'))\,z\,(\mathsf{mk}\widehat{\mathsf{Mu}}^{-1}\,mu)\Big)\Big).$$

Uniqueness is obtained from well-founded induction on sizes and the functor laws for F.

4.2.3 Initial Algebras. Finally we define the initial algebra as $\mathsf{Mu} :\equiv \lambda z.\exists n.\widehat{\mathsf{Mu}}\,n\,z$. However, in order to define the algebra map $\mathsf{mkMu} : \Pi(z : Z).F\,\mathsf{Mu}\,z \to \mathsf{Mu}\,z$ we need an extra property of F: we say that F *weakly commutes with* $\exists(n : \mathsf{Size})$ if the canonical map of type $\Pi(z : Z).(\exists n.\hat{F}\,A\,n\,z) \to F(\lambda z'.\exists n.A\,n\,z')\,z$ is an isomorphism for every A. All finitely branching indexed containers [Altenkirch et al. 2006], i.e. functors of the form $F\,X\,z \equiv \Sigma(c : C\,z).(\Pi(b : B\,z\,c).X\,(r\,z\,c\,b))$ for some $r : \Pi(z : Z).\Pi(c : C\,z).B\,z\,c \to Z$ with $B\,z\,c$ finite, satisfy this property. If F weakly commutes with $\exists(n : \mathsf{Size})$, then $F\,\mathsf{Mu} \cong \mathsf{Mu}$, which gives us the algebra map as anticipated. The unique algebra morphism fold is defined using $\widehat{\mathsf{fold}}$ and its uniqueness follows from the fact that there is an isomorphism between algebra morphisms from $(\mathsf{Mu}, \mathsf{mkMu})$ and algebra morphisms from $(\widehat{\mathsf{Mu}}, \mathsf{mk}\widehat{\mathsf{Mu}})$.

4.2.4 Final Co-algebras. The entire construction can be dualized to obtain final co-algebras, here we give only the main definitions:

$$\hat{F} :\equiv \lambda A.\lambda n.F\,(\lambda z'.\forall m < n.A\,m\,z),$$
$$\widehat{\mathsf{Nu}} :\equiv \mathsf{fix}\,(\lambda n.\lambda \mathsf{Nu}'.F(\lambda z'.\forall(m : \mathsf{Size}).\Pi(e : m < n).\mathsf{Nu}'\,m\,e\,y),$$
$$\mathsf{Nu} :\equiv \lambda z.\forall n.\widehat{\mathsf{Nu}}\,n\,z.$$

Then Nu is the final co-algebra if F weakly commutes with $\forall(n : \mathsf{Size})$, i.e. if the canonical map of type $\Pi(z : Z).F(\lambda z'.\forall(n : \mathsf{Size}).A\,n\,z')\,z \to \forall n.\hat{F}\,A\,n\,z$ is an isomorphism for every A. This property is satisfied by all containers.

5 SOUNDNESS: AN OVERVIEW OF THE PRESHEAF MODEL

In this section we give a high-level overview of the presheaf model that we constructed to prove the soundness of ParamDTT. This section can be safely skipped by readers who are primarily interested in the type theory side of the story. No prior knowledge of presheaves is assumed.

We start with a brief review of the set model of MLTT and show why it cannot model contexts containing continuous or parametric interval variables and hence also fails to model bridges and paths. We then explain how the general presheaf model of dependent type theory as treated by Hofmann [1997], can be seen as a method for manually adding non-setlike 'primitive' contexts to the set model. We point out existing presheaf models that support some form of interval: the reflexive graph model [Atkey et al. 2014] and cubical models [Bernardy et al. 2015; Bezem et al. 2014; Cohen et al. 2016]. Finally, we construct a specific presheaf model by manually adding *bridge/path cubes* — contexts consisting solely of continuous and parametric interval variables — and give a high-level overview of the semantics of ParamDTT in that model.

For the sake of the exposition, we ignore universe level issues. Most of the time, we do not use an interpretation function but instead give judgements a direct meaning in the model. A more formal treatment of the model is found in [Nuyts 2017].

5.1 The Set Model of Dependent Type Theory

In this model, a closed type is a set, whose elements are precisely its semantic closed terms. So $\vdash T$ type means that T is a set, and $\vdash t : T$ means that $t \in T$. A context Γ is modelled as a set, whose elements are vectors of semantic closed terms that give meaning to all of the variables in Γ:

$$() = \{()\}, \qquad (\Gamma, t : T) = \{(\gamma, t) \mid \gamma \in \Gamma, t \in T[\gamma]\}.$$

So $\Gamma \vdash \mathrm{Ctx}$ means that Γ is a set. From the notation $T[\gamma]$, it is already clear that an open type $\Gamma \vdash T$ type is a function that assigns to every $\gamma \in \Gamma$ a set $T[\gamma]$. A term $\Gamma \vdash t : T$ is a function that maps $\gamma \in \Gamma$ to $t[\gamma] \in T[\gamma]$. Note that elements of a context Γ correspond precisely to semantic substitutions $\gamma : () \to \Gamma$ from the empty context. More generally, a substitution $\sigma : \Delta \to \Gamma$ is a function from Δ to Γ. Substitutions of types and terms are given by $T[\sigma][\delta] = T[\sigma(\delta)]$ and $t[\sigma][\delta] = t[\sigma(\delta)]$. Definitional equality is modelled as equality of mathematical objects.

So to wrap up, we have picked a category (namely the category of sets) whose objects model contexts and whose morphisms model substitutions. We have defined for every object Γ what it means to be a type $\Gamma \vdash T$ type and how we define $T[\sigma]$, and finally we have done the same for terms. We have explained how to extend a context with a type. A few additional features will give this setup the structure of a *category with families* [CwF, Dybjer 1996], and we can prove soundness of MLTT with any extensions of interest by giving every inference rule a meaning in the model in such a way that some contradictory type has no semantic terms.

The set model is insufficient to model ParamDTT. Indeed, how do we interpret the closed type $\mathbb{I}$ or the contexts $(i : \mathbb{I})$ and $(i^\sharp : \mathbb{I})$ as a set? The terms $\vdash 0, 1 : \mathbb{I}$ show that these have at least two elements, and $(0 \doteq 1) \equiv \bot$ shows that they are distinct. But there is nothing that allows us to relate them, which is necessary if we want to model the path degeneracy axiom. The interval is, in short, not a set.

5.2 Presheaf Models

The problem of contexts that cannot be modelled by sets, can be overcome by adding them explicitly to the model. This is what presheaf models do. The first step in constructing a presheaf model, is to identify a set of *primitive contexts* (also called levels, shapes, worlds or simply objects) that 'generate' the problem. Secondly, for every two primitive contexts V, W, we must explicitly provide the set of all substitutions $\varphi : V \xrightarrow{\mathrm{p}} W$ that we want to exist between V and W; we will call these

primitive substitutions (more commonly, they are called face or restriction maps). Together, these must form a category $\mathcal{W}$ (the base category).

Now, we define contexts Γ not in relation to the empty context by providing the set of all substitutions $() \to \Gamma$, but in relation to all primitive contexts. So in order to define Γ, we need to provide for every primitive context W the set of all *defining substitutions* $\gamma : W \xrightarrow{\mathrm{D}} \Gamma$ *and* for every primitive substitution $\varphi : V \xrightarrow{\mathrm{P}} W$ a composition operator $\llcorner \circ \varphi : (W \xrightarrow{\mathrm{D}} \Gamma) \to (V \xrightarrow{\mathrm{D}} \Gamma)$. In other words, a context Γ is a functor $(\llcorner \xrightarrow{\mathrm{D}} \Gamma) : \mathcal{W}^{\mathrm{op}} \to \mathrm{Set}$ (also called a presheaf over $\mathcal{W}$). A substitution $\sigma : \Delta \to \Gamma$ (also: presheaf map/morphism) is then defined by saying how it composes with the defining substitutions of Δ, i.e. we have to give an operator $\sigma\circ : (W \to \Delta) \to (W \to \Gamma)$ that is natural in W. In other words, substitutions are natural transformations, and our category of contexts is the functor space $\mathrm{Set}^{\mathcal{W}^{\mathrm{op}}}$, also called $\widehat{\mathcal{W}}$.

Note that we can view a primitive context W as a context by setting $(V \xrightarrow{\mathrm{D}} W) := (V \xrightarrow{\mathrm{P}} W)$, and a primitive substitution $\varphi : V \xrightarrow{\mathrm{P}} W$ as a substitution by defining $\varphi \circ \llcorner$ as composition with φ in $\mathcal{W}$. This defines a functor $\mathrm{y} : \mathcal{W} \to \widehat{\mathcal{W}}$ called the *Yoneda-embedding*, which is fully faithful, meaning that $(V \xrightarrow{\mathrm{P}} W) \cong (V \to W)$, i.e. every substitution between primitive contexts is in fact a primitive substitution. Similarly, we have $(W \xrightarrow{\mathrm{D}} \Gamma) \cong (W \to \Gamma)$.

So $\Gamma \vdash \mathrm{Ctx}$ means that Γ is a presheaf, and a substitution $\sigma : \Delta \to \Gamma$ is a presheaf map. We define a type $\Gamma \vdash T$ type by giving for every W and every $\gamma : W \xrightarrow{\mathrm{D}} \Gamma$ its set of *defining terms* $W \vdash^{\mathrm{D}} t : T[\gamma]$, plus for every $\varphi : V \xrightarrow{\mathrm{P}} W$ a substitution operator $\llcorner[\varphi]$ that takes $W \vdash^{\mathrm{D}} t : T[\gamma]$ to $V \vdash^{\mathrm{D}} t[\varphi] : T[\gamma \circ \varphi]$. A term $\Gamma \vdash t : T$ is then a thing that maps defining substitutions $\gamma : W \xrightarrow{\mathrm{D}} \Gamma$ to defining terms $W \vdash^{\mathrm{D}} t[\gamma] : T[\gamma]$ in such a way that $t[\gamma][\varphi] = t[\gamma \circ \varphi]$. Again, one can show that defining terms $W \vdash^{\mathrm{D}} t : T[\mathrm{id}]$ are in bijection to terms $W \vdash t : T$. We extend contexts as follows:

$$W \xrightarrow{\mathrm{D}} (\Gamma, t : T) = \left\{ (\gamma, t) \,\middle|\, \gamma : W \xrightarrow{\mathrm{D}} \Gamma \text{ and } W \vdash^{\mathrm{D}} t : T[\gamma] \right\}, \qquad (\gamma, t) \circ \varphi = (\gamma \circ \varphi, t[\varphi]).$$

One can show that every presheaf category constitutes a CwF [Hofmann 1997] that supports universes if the metatheory does [Hofmann and Streicher 1997].

The category of reflexive graphs. In order to model parametricity for a dependent type system without modalities, we can pick as base category the category RG with just two primitive contexts $()$ and $(i : \mathbb{I})$, and where primitive substitutions are (non-freely) generated by $(0/i), (1/i) : () \to (i : \mathbb{I})$ and $() : (i : \mathbb{I}) \to ()$. A presheaf Γ over RG is then a reflexive graph, with a set of nodes $() \xrightarrow{\mathrm{D}} \Gamma$ and a set of edges $(i : \mathbb{I}) \xrightarrow{\mathrm{D}} \Gamma$. Precomposition with $(0/i)$ and $(1/i)$ determines the source and target of an edge, and precomposition with $()$ determines the reflexive edge at a node. This corresponds to the reflexive graph model as treated by Atkey et al. [2014], although they use a non-standard universe to model identity extension (IEL). This model does not support iterated parametricity; hence it does not support internal parametricity operators which, in absence of modalities, can be self-applied. The non-standard universe supports IEL, but not in combination with proof-relevant relations.

The category of cubical sets. In order to support iterated parametricity or to have identity extension in the presence of proof-relevant relations, we need a way to express relatedness of relations. In other words, we need a notion of edges between edges (squares), edges between squares (cubes), etc. Although contexts like $(i : \mathbb{I}, j : \mathbb{I})$ exist in the category of reflexive graphs, these are still just graphs that do not contain data to express that the faces $(0/j), (1/j) : (i : \mathbb{I}) \xrightarrow{\mathrm{D}} (i : \mathbb{I}, j : \mathbb{I})$ are related.

In the cubical model, we add cubes explicitly. As base category, we pick the *cube category* Cube with primitive contexts $(\vec{i} : \mathbb{I}^n)$ for $n \in \mathbb{N}$ and primitive substitutions $(\vec{i} : \mathbb{I}^m) \xrightarrow{\mathrm{P}} (\vec{i} : \mathbb{I}^n)$ that substitute every variable of the codomain with either 0, 1 or a variable from the domain. A presheaf Γ over Cube is then a so-called cubical set, consisting of, for every number n, a set of n-dimensional

cubes $(\vec{i} : \mathbb{I}^n) \xrightarrow{\mathrm{D}} \Gamma$ and a composition operator with primitive substitutions (face maps) that allow us to extract faces, diagonals and reflexive cubes from a given cube. This model is close to the (binary version of) the one used by Bernardy et al. [2015]. Cubical type theory [Bezem et al. 2014; Cohen et al. 2016] uses a similar model to model univalence, but they have additional operators $\vee$, $\wedge$ and $\neg$ on the interval, resulting in a base category with the same objects but more primitive substitutions.

The category of bridge/path cubical sets. We need to make just one modification to the base category to obtain a presheaf category of our system: we annotate the interval variables in primitive contexts with either **id** or $\sharp$ (as $(i^{\P} : \mathbb{I})$ really just contains two points 0 and 1). So as base category, we pick the category BPCube whose objects are *bridge/path cubes* $(\vec{j} : \mathbb{I}^m, \vec{i}^{\sharp} : \mathbb{I}^n)$, where $m, n \in \mathbb{N}$. Primitive substitutions $\varphi : V \longrightarrow W$ substitute every continuous 'bridge' variable from W with 0, 1 or a bridge variable from V, and every parametric 'path' variable from W with 0, 1 or any variable from V (Fig. 7). A presheaf Γ is then a bridge/path cubical set that contains for any $m, n \in \mathbb{N}$ a set of cubes with m bridge dimensions and n path dimensions. The defining composition operator of Γ allows us to extract faces, diagonals and reflexive cubes from a given cube, as well as to weaken path dimensions to bridge dimensions.

The standard presheaf model over BPCube supports iterated parametricity as it is essentially just the cubical set model extended with an arbitrary distinction between bridge and path dimensions. However, on top of this model we will build a machinery of modalities that adds the path degeneracy axiom, but loses full internal iterated parametricity. We still need the cubical structure in order to support the path degeneracy axiom in the presence of proof-relevant relations.

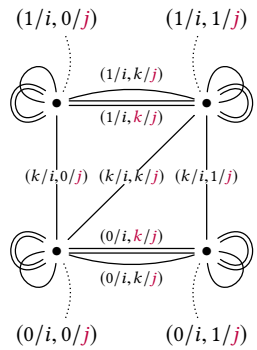

Fig. 7. All points, bridges (single line) and paths (double line) of the bridge/path cube $(i : \mathbb{I}, j^{\sharp} : \mathbb{I})$. The labels are the corresponding primitive substitutions with domains $()$, $(k : \mathbb{I})$ and $(k^{\sharp} : \mathbb{I})$ respectively.

5.3 The Cohesive Structure of $\widehat{\text{BPCube}}$

A bridge/path cubical set $\Gamma \in \widehat{\text{BPCube}}$ can be seen as an ordinary cubical set (namely the cubical set of its bridges) equipped with a notion of *cohesion* in the sense of Licata and Shulman [2016] expressed by its paths. This is formalized by a *forgetful* functor $\sqcup : \widehat{\text{BPCube}} \longrightarrow \widehat{\text{Cube}}$ that forgets paths. This functor can be shown to be a morphism of CwFs, i.e. it extends to types and terms in a well-behaved manner. It is part of a chain of at least five adjoint functors (Fig. 8):

$$\sqcap \dashv \triangle \dashv \sqcup \dashv \nabla \dashv \boxminus, \qquad \sqcap, \sqcup, \boxminus : \widehat{\text{BPCube}} \longrightarrow \widehat{\text{Cube}}, \qquad \triangle, \nabla : \widehat{\text{Cube}} \longrightarrow \widehat{\text{BPCube}}$$

of which only $\sqcap$ is *not* a morphism of CwFs. The *discrete* functor $\triangle$ takes a cubical set with only bridges and equips it with a path relation defined as the equality relation — the strictest path relation possible. The *codiscrete* functor ∇ on the other hand defines the path relation as the bridge relation — the most liberal possibility. The functor $\boxminus$ is a forgetful functor at another level: it maps a bridge/path cubical set to its cubical set of paths and forgets the bridge structure. From this perspective, ∇ takes a cubical set with only paths and equips it with a bridge relation defined as the path relation — the strictest possibility. Finally, the functor $\sqcap$ identifies all path-connected objects, producing a cubical set with only a bridge relation. This functor is not a morphism of CwFs as it would not respect substitution.

Composing each adjoint pair to a (co)monad $\widehat{\text{BPCube}} \longrightarrow \widehat{\text{BPCube}}$, we get a chain of four adjoint endofunctors on $\widehat{\text{BPCube}}$,

$$\int \dashv \flat \dashv \sharp \dashv \P : \widehat{\mathsf{BPCube}} \to \widehat{\mathsf{BPCube}}.$$

The *shape* monad $\int = \triangle\sqcap$, again the only one that is not a morphism of CwFs, identifies path-connected objects and then reintroduces a discrete path relation. The *flat* co-monad $\flat = \triangle\sqcup$ redefines the path relation discretely as the equality relation. The *sharp* monad $\sharp = \triangledown\sqcup$ redefines the path relation codiscretely as the bridge relation, and the *pointwise* co-monad $\P = \triangledown\boxminus$ redefines the bridge relation as the path relation. Note that a presheaf map $\sharp\Delta \to \Gamma$ maps bridges in Δ to paths in Γ, as does a parametric function, and that a presheaf map $\P\Delta \to \Gamma$ maps paths in Δ to paths in Γ, but does not act on bridges in Δ, just like a pointwise function. The functors $\P$, Id and $\sharp$ satisfy the same composition rules that we have for modalities, and we have the co-unit $\vartheta : \P \to \mathrm{Id}$ and the unit $\iota : \mathrm{Id} \to \sharp$ to interpret the order relation on modalities.

5.4 Meaning of Judgements

In this section, we write internal judgements between interpretation brackets $[\![\ldots]\!]$, in order to tell them apart from mathematical statements about the standard presheaf model from Section 5.2 which we also write in judgement-style.

Contexts. As usual, $[\![\Gamma \vdash \mathsf{Ctx}]\!]$ means that $[\![\Gamma]\!]$ is a presheaf over BPCube, i.e. $[\![\Gamma]\!] \vdash \mathsf{Ctx}$.

Types. Unusually, $[\![\Gamma \vdash T\ \mathsf{type}]\!]$ means that $[\![T]\!]$ is a type over the presheaf $\sharp[\![\Gamma]\!]$, i.e. $\sharp[\![\Gamma]\!] \vdash [\![T]\!]$ type, that is moreover *discrete*. The fact that it lives in $\sharp[\![\Gamma]\!]$ means that when there is a bridge in $[\![\Gamma]\!]$, then $[\![T]\!]$ provides a notion of paths over it. This is necessary if we want to add $x^{\P} : T$ to the context, because a bridge in $\P[\![T]\!]$ is a path in $[\![T]\!]$. Of course we have $[\![\Gamma]\!] \vdash [\![T]\!][\iota]$ type. We say that a type $\Delta \vdash A$ type is *discrete* if every path in A that lives above a constant path in Δ, is also constant. More formally, if we have a primitive context $(W, k^{\sharp} : \mathbb{I})$ and a defining substitution $\delta : (W, k^{\sharp} : \mathbb{I}) \xrightarrow{\mathrm{D}} \Delta$ that is constant in k, i.e. it factors as $(W, k^{\sharp} : \mathbb{I}) \xrightarrow{0} W \xrightarrow{\delta'} \Delta$, then every defining term $(W, k^{\sharp} : \mathbb{I}) \vdash^{\mathrm{D}} a : A[\delta]$ is also constant in k, i.e. it is a weakening of some $W \vdash^{\mathrm{D}} a' : A[\delta']$. This is essentially the statement that non-dependent paths in A are constant, i.e. that A satisfies the path degeneracy axiom. So $[\![\Gamma \vdash T\ \mathsf{type}]\!]$ means that $[\![T]\!]$ is a type in context $\sharp[\![\Gamma]\!]$ that satisfies the path degeneracy axiom.

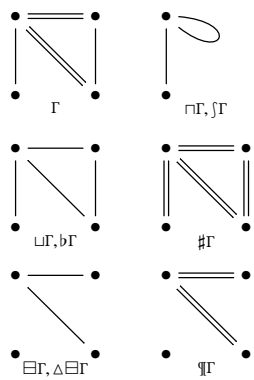

Fig. 8. Example of a bridge/path cubical set Γ and its images for various functors. A single line denotes an edge (for edge cubical sets) or a bridge, a double line is a path. Implied content, such as constant lines and the bridge under a path, are not shown. Hence, applying $\triangle$ preserves the drawing.

Modal context extension. If $[\![\Gamma \vdash T\ \mathsf{type}]\!]$ holds, then we need to be able to extend $[\![\Gamma]\!]$ with variables of type $[\![T]\!]$ in any modality μ. We already know that μ corresponds to an endomorphism of CwFs on $\widehat{\mathsf{BPCube}}$. The laws of a morphism of CwFs allow us to apply μ to both sides of a typing judgement, obtaining $\mu\sharp[\![\Gamma]\!] \vdash \mu[\![T]\!]$ type. Now $\mu\sharp = \sharp$ by the composition table of modalities so that we have $[\![\Gamma]\!] \vdash (\mu[\![T]\!])[\iota]$ type (note how this would not work if $[\![T]\!]$ lived in $[\![\Gamma]\!]$ and $\mu = \P$). We now interpret $[\![\Gamma, x^{\mu} : T]\!]$ as $([\![\Gamma]\!], x : (\mu[\![T]\!])[\iota])$. Then one can show that $[\![\sharp \setminus \Gamma]\!] \cong \flat[\![\Gamma]\!]$ and $[\![\P \setminus \Gamma]\!] = \sharp[\![\Gamma]\!]$, so that left dividing a context by a modality corresponds to applying the modality's left adjoint.

Terms. A term $[\![\Gamma \vdash t : T]\!]$ is now interpreted as $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!][\iota]$.

5.5 Some Remarkable Interpretations of Types

The universe. For simplicity, we ignore universe levels — a more formal exposition can be found in [Nuyts 2017]. The standard presheaf universe $\mathcal{U}^{\mathrm{Psh}}$ as described by Hofmann and Streicher

[1997] has as its defining terms $W \vdash^{\mathrm{D}} A : \mathcal{U}^{\mathrm{Psh}}$ the types $W \vdash A\,\mathrm{type}$ over the Yoneda-embedded primitive context W. This universe is problematic for our purposes in three ways. First, the types it contains are not all discrete. Secondly, $\mathcal{U}^{\mathrm{Psh}}$ is not discrete itself and thus cannot satisfy the path degeneracy axiom. Thirdly, an encoded type $\Gamma \vdash A : \mathcal{U}$ should admit terms $\sharp\Gamma \vdash a : A$; indeed, the $\forall$-quantifier admits parametric functions in non-parametric types. The universe $\mathcal{U}^{\mathrm{Psh}}$ does not satisfy this requirement.

The first issue is easiest to resolve: we define a non-discrete universe of discrete types $\mathcal{U}^{\mathrm{NDD}}$ by taking as defining terms $W \vdash^{\mathrm{D}} A : \mathcal{U}^{\mathrm{NDD}}$ precisely the *discrete* types $W \vdash A\,\mathrm{type}$.

The other two issues can be solved together. A closed type such as the universe that we want to model, is discrete if and only if its path relation is the equality relation. Meanwhile, a bridge $(i : \mathbb{I}) \vdash^{\mathrm{D}} A : \mathcal{U}$ should admit paths $(i^\sharp : \mathbb{I}) \vdash^{\mathrm{D}} a : A$. Both of these matters are resolved if we take as defining terms $(\vec{j} : \mathbb{I}^m, \vec{i}^\sharp : \mathbb{I}^n) \vdash^{\mathrm{D}} A : \mathcal{U}$ precisely the discrete types $(\vec{j}^\sharp : \mathbb{I}^m) \vdash A\,\mathrm{type}$, or equivalently terms $(\vec{j}^\sharp : \mathbb{I}^m) \vdash^{\mathrm{D}} A : \mathcal{U}^{\mathrm{NDD}}$. So the paths of $\mathcal{U}$ are constant and the bridges of $\mathcal{U}$ are the paths of $\mathcal{U}^{\mathrm{NDD}}$; this amounts to saying that $\mathcal{U} = \flat\P\mathcal{U}^{\mathrm{NDD}} = \triangle\boxminus\mathcal{U}^{\mathrm{NDD}}$. Indeed: $\P$ redefines the bridge relation as the path relation, and $\flat$ subsequently redefines the path relation as equality.

Now if we have $[\![\sharp \setminus \Gamma \vdash A : \mathcal{U}]\!]$, i.e. (up to isomorphism) $\flat[\![\Gamma]\!] \vdash [\![A]\!] : \flat\P\mathcal{U}^{\mathrm{NDD}}$, then applying $\sharp$ and using that $\sharp\flat = \sharp$ and $\sharp\P = \P$ yields a term $\sharp[\![\Gamma]\!] \vdash {}^\sharp[\![A]\!] : \P\mathcal{U}^{\mathrm{NDD}}$. Finally we can use $\vartheta : \P \to \mathrm{Id}$ to obtain $\sharp[\![\Gamma]\!] \vdash \vartheta({}^\sharp[\![A]\!]) : \mathcal{U}^{\mathrm{NDD}}$, which encodes a discrete type $\sharp[\![\Gamma]\!] \vdash \vartheta({}^\sharp[\![A]\!])\,\mathrm{type}$, i.e. an interpretation for $[\![\Gamma \vdash A\,\mathrm{type}]\!]$. We will write $\mathsf{El}\,T$ for $\vartheta({}^\sharp T)$.

An important challenge is now to establish that all type formers work on elements of the discrete universe $\mathcal{U} = \flat\P\mathcal{U}^{\mathrm{NDD}}$. The general approach is the following: We move the functor $\flat\P$ to the left of the turnstile ($\vdash$) in the form of its left adjoint $\sharp\int$. Then we apply existing type formers for $\mathcal{U}^{\mathrm{NDD}}$ in the context $\sharp\int\Gamma$ and finally we move the functor back to the right. Great care is needed where we have interactions between terms and types, i.e. when modelling dependent quantifiers, identity types and Glue/Weld types.

Identity types. The standard presheaf identity type has a single defining term $W \vdash^{\mathrm{D}} \star : (a =_A b)[\gamma]$ when $a[\gamma] = b[\gamma]$ and no defining terms for γ otherwise. In other words, it represents definitional, proof-irrelevant equality. Now suppose we have $[\![\Gamma \vdash T : \mathcal{U}]\!]$ and $[\![\Gamma \vdash s, t : T]\!]$. The former judgement can be unfolded to $\sharp\int[\![\Gamma]\!] \vdash T'\,\mathrm{type}$. Using the fact that T is discrete, we are allowed to disregard the $\int$ functor, so by applying $\sharp$ to $[\![s]\!]$ and $[\![t]\!]$, we obtain terms $\sharp\int[\![\Gamma]\!] \vdash s', t' : \sharp T'$. Then we can construct $\sharp\int[\![\Gamma]\!] \vdash s' =_{\sharp T'} t'\,\mathrm{type}$, which is discrete as it is a proposition, and finally we can go back to $[\![\mathcal{U}]\!]$. It is noteworthy that the interpretation of the identity type is not over T', but over $\sharp T'$. This also allows us to model parametricity of the reflexivity constructor, without damaging the power of the J-rule.

Function types. Assume we have $[\![\Gamma \vdash A : \mathcal{U}]\!]$ and $[\![\Gamma, x^{\sharp\setminus\mu} : A \vdash B : \mathcal{U}]\!]$, i.e. $[\![\Gamma]\!] \vdash [\![A]\!] : \flat\P\mathcal{U}^{\mathrm{NDD}}$ and $[\![\Gamma]\!], x : ((\sharp \setminus \mu)\mathsf{El}\,A)[\iota] \vdash B : \flat\P\mathcal{U}^{\mathrm{NDD}}$. Again, we push $\flat\P$ to the left in the form of $\sharp\int$. Remember that $\int$ is a monad that we can map *into*; this allows us to extract the variable x out of it. Meanwhile, we have a natural transformation $\mu \to \sharp \circ (\sharp \setminus \mu)$. All of this allows us to derive $\sharp\int[\![\Gamma]\!] \vdash A' : \mathcal{U}^{\mathrm{NDD}}$ and $\sharp\int[\![\Gamma]\!], x : \mu A' \vdash B' : \mathcal{U}^{\mathrm{NDD}}$, whence $\sharp\int[\![\Gamma]\!] \vdash \Pi(x : \mu A').B' : \mathcal{U}^{\mathrm{NDD}}$ and finally $[\![\Gamma]\!] \vdash [\![\Pi^\mu(x : A).B]\!] : \flat\P\mathcal{U}^{\mathrm{NDD}}$.

Pair types. The pointwise and continuous pair types are formed in the same way as the function types. For the parametric pair type $\exists$, a problem occurs because $\Sigma(x : \sharp A').B'$ is not generally discrete. We resolve this by applying a type-level shape operator $\oint$ that identifies all terms connected by non-dependent paths, obtaining $\sharp\int[\![\Gamma]\!] \vdash \oint\Sigma(x : \sharp A').B' : \mathcal{U}^{\mathrm{NDD}}$. Pushing $\sharp\int$ to the right again, we find $[\![\Gamma]\!] \vdash [\![\exists(x : A).B]\!] : \flat\P\mathcal{U}^{\mathrm{NDD}}$.

Face predicates. Similar to $\mathcal{U}^{\mathsf{Psh}}$, there is a standard presheaf universe of propositions $\mathsf{Prop}^{\mathsf{Psh}}$. The defining terms $W \vdash^{\mathrm{D}} P : \mathsf{Prop}^{\mathsf{Psh}}$ are the *proof-irrelevant* types $W \vdash P$ type over the Yoneda-embedded primitive context W; these are types for which all defining terms are equal to $\star$. We define $[\![\mathbb{F}]\!] = \flat\P(\sharp\mathsf{Prop}^{\mathsf{Psh}}) = \flat\mathsf{Prop}^{\mathsf{Psh}}$ (so $\sharp\mathsf{Prop}^{\mathsf{Psh}}$ takes the role that $\mathcal{U}^{\mathsf{Psh}}$ took before) and interpret context extension similar to the way we did for types. See [Nuyts 2017] for details.

Glueing and welding. We first define these types in the standard presheaf model. Write $W = \mathsf{Weld}\{A \to (P\,?\,T, f)\}$ and $G = \mathsf{Glue}\{A \leftarrow (P\,?\,T, h)\}$. We want these to be types: $\Gamma \vdash W, G$ type. We define W as follows: the defining terms $V \vdash^{\mathrm{D}} w : W[\gamma]$ are defining terms $V \vdash^{\mathrm{D}} w : T[(\gamma, \star)]$ if $P[\gamma] = \top$; otherwise, they are defining terms $V \vdash^{\mathrm{D}} w : A[\gamma]$. The primitive substitutions $w[\varphi]$ of w are defined using some case distinctions and the function f. Similarly, a defining term $V \vdash^{\mathrm{D}} g : G[\gamma]$ is a defining term $V \vdash^{\mathrm{D}} g : T[(\gamma, \star)]$ if $P[\gamma] = \top$. Otherwise, it is a pair (a, t) with $V \vdash^{\mathrm{D}} a : A[\gamma]$ and $V, p : P \vdash t : T[(\gamma, p)]$, such that h maps t to a. Again, primitive substitutions are defined using case distinction and the function h.

The next step is to internalize these types. We have $[\![A]\!]$ and $[\![T]\!]$ living in $\flat\P\mathcal{U}^{\mathsf{NDD}}$ and $[\![P]\!]$ living in $\flat\P\sharp\mathsf{Prop}^{\mathsf{Psh}}$, all in context $[\![\Gamma]\!]$. In each of these cases, we can push $\flat\P$ to the left, arriving in context $\sharp\int[\![\Gamma]\!]$. Further unpacking is needed for $[\![P]\!]$. Meanwhile, $[\![f]\!]$ lives in context $[\![\P \setminus \Gamma]\!] \cong \sharp[\![\Gamma]\!]$. The fact that the type of f is discrete and lives in a discrete universe, allows us to disregard $\int$, so that f too lives in $\sharp\int[\![\Gamma]\!]$. Now we can apply the presheaf Glue or Weld type formers and push the functors to the right again.

Size. We define a closed type Size in the presheaf model as a type of naturals with codiscrete bridge structure and discrete path structure: a defining term $(\vec{j} : \mathbb{I}^m, \vec{i}^{\sharp} : \mathbb{I}^n) \vdash^{\mathrm{D}} s : \mathsf{Size}$ consists of 2^m natural numbers, one for each vertex of the bridge cube. So we relate any two numbers with a bridge, while paths are required to be constant. The type $\sharp\mathsf{Size}$ then has defining terms $(\vec{j} : \mathbb{I}^m, \vec{i}^{\sharp} : \mathbb{I}^n) \vdash^{\mathrm{D}} s : \sharp\mathsf{Size}$ consisting of 2^{m+n} natural numbers. Given $\Gamma \vdash s, t : \sharp\mathsf{Size}$, we define a proposition $\Gamma \vdash s \leq t$ type, where we have a defining term $W \vdash^{\mathrm{D}} \star : (s \leq t)[\gamma]$ if and only if $s[\gamma] \leq t[\gamma]$ vertexwise. The internal type $[\![s \leq t]\!]$ is then defined similarly to the identity type. Finally, the fixpoint operator on Size is defined by induction on the greatest vertex of a given cube.

6 RELATED AND FUTURE WORK

Parametricity in and of dependent type theory. Figure 9 provides an overview of related work. Reynolds's original formulation [1983] is in terms of a (partly ill-conjectured [Reynolds 1984]) set-theoretic semantics, but others have shown that parametricity of System F can be formulated in a predicate logic on System F. Such frameworks can be seen as 'internal' because the parametricity proof that we obtain for a specific program, can be constructed as a proof term internally in the predicate logic. However, as the 'journey' column emphasizes, the fact that this program-to-proof translation works for *any* program, is proven externally. IEL holds in each of the cited frameworks for System F.

The second chunk of Fig. 9 contains frameworks for parametricity of dependent type theory. In this class of work, IEL is only shown to hold for small types. In fact, the function leak from Section 1 would be ruled out by IEL; hence, IEL cannot hold in general. In our terminology, work like Bernardy et al.'s [2012] which omits IEL altogether, shows that MLTT is *continuous* (i.e. all functions respect relations) analogous to Lemma 3.5 where μ is the continuous modality.

Atkey et al. [2014] prove their results in a reflexive graph model, following related models for simpler type systems such as those by Robinson and Rosolini [1994], Hasegawa [1994], and Atkey [2012]. This model has been enhanced by Bernardy et al. [2015] to a (unary, but generalizable) model in terms of cubical sets (iterated reflexive graphs) that supports iterated parametricity. Our work builds further on that. Bernardy et al.'s type system provides operators that allow us to map

citation	source	target	journey	model	IEL proof
[Reynolds 1983]	System F			conject. set model	yes
[Abadi et al. 1993]	System F	System $\mathscr{R}$	external		yes
[Plotkin and Abadi 1993]	System F	System F + logic	external		yes
[Wadler 2007]	System F	System F + logic	external		yes
[Takeuti 2001]	$\mathscr{X} \in \lambda$-cube	$\mathscr{Y} \in \lambda$-cube	external		for small types
[Bernardy et al. 2012]	any PTS	other PTS	external		no
[Krishnaswami and Dreyer 2013]	dependent types			PER-model	only some corollaries
[Atkey et al. 2014]	dependent types			presheaves: reflexive graphs	for small types
[Bernardy et al. 2015]	dependent types + param. operators	same as source	internal	presheaves: (unary) cubical sets	no
This work	dependent types + Glue, Weld, $\forall$, $\exists$	same as source	internal	presheaves: bridge/path cubical sets	yes: (semantics of) degax

Fig. 9. Classification of important related frameworks that prove parametricity. The 'source' column lists the type system that parametricity is proven of. If parametricity is formulated in some type system, it is listed under 'target' and the 'journey' column lists whether the translation from program to parametricity proof, takes place internal to the type system, or externally in the metatheory. If a metatheoretic model is (also) used, it is listed under 'model'. The last column lists whether the identity extension lemma (IEL) is proven.

programs to their parametricity proofs *internal* to the type system. Modulo some technical issues that can be overcome, their operators could be plugged into our presheaf model and supplementing them with the path degeneracy axiom would have given our system similar power. However, just as in their system, we would have had non-duplicable interval variables, severely complicating implementation as an extension of Agda. We overcome this, somewhat experimentally, by choosing instead to use the more indirect Glue and Weld types, which exist in any presheaf model and are in this sense also more robust against future reworkings of the model. This decision is unrelated to the appearance of the pointwise modality (¶). The graph type $/f\backslash$ from Section 3.2.1 with its push and pull functions, is analogous to the graph relation in System $\mathscr{R}$ [Abadi et al. 1993].

Modalities. Although the use of modalities for keeping track of parametricity is to our knowledge new, parametricity is just one addition to a large list of applications of modalities, including (eponymously) modal logic [Pfenning and Davies 2001], variance of functorial dependencies [Abel 2006, 2008; Licata and Harper 2011], irrelevance [Abel and Scherer 2012; Reed 2003], erasure [Mishra-Linger and Sheard 2008], intensionality vs. extensionality [Pfenning 2001]. Licata and Shulman's modality system for axiomatic cohesion [2016] is an important ingredient of our model. The syntactic treatment in terms of order, composition and left division has been developed by Pfenning [2001] and Abel [2006, 2008], and was already implemented in Agda as the basis for its irrelevance modality, facilitating the implementation of the ParamDTT extension of Agda.

Parametricity versus irrelevance. Notions closely related to parametricity, especially in non-dependently-typed systems, are irrelevance and erasability. The meanings of these words seem to shift somewhat throughout the literature, so we start by defining the terminology we will use. By a *parametric* dependency, as in the rest of the paper, we mean a dependency that maps related inputs to (heterogeneously) equal outputs. This includes the identity function $\sharp\mathsf{Nat} \to \mathsf{Nat}$ defined in Eq. (2). By an *erasable* dependency, we mean a dependency that can be erased after type-checking, at compile time, while preserving the operational semantics of a program. By an *irrelevant* dependency, we mean a dependency that can be erased already during type-checking, implying that terms can be converted between types that are equal up to their irrelevant parts. It is intuitively clear that irrelevance is stronger than erasability, which in turn is stronger than parametricity.

Mishra-Linger and Sheard's EPTS [2008] and Barras and Bernardo's ICC* [2008] are type systems with quantifiers for erasable dependencies based on Miquel's implicit calculus of constructions (ICC) [2001a; 2001b]. Both propose a conversion rule that erases at type-checking time, making their quantifiers irrelevant. If we allow conversion only between β-equal types, as Mishra-Linger and Sheard also suggest, both systems embed into ours. Abel and Scherer [2012] show that it is problematic to view the quantifiers of EPTS and ICC* as irrelevant. The problem can be neatly formulated in terms of our bridges and paths: if a function is to be irrelevant, then surely it must map any pair of inputs to equal outputs. However, both type systems consider irrelevant functions $f : \Pi^{\mathsf{irr}}(x : A).B$ with merely 'continuous' codomain $B : A \to \mathcal{U}$. In our system, this means that we would require a path between any two values $f\ a_1^{\mathsf{irr}}$ and $f\ a_2^{\mathsf{irr}}$ without necessarily providing a *notion* of paths between $B\,a_1$ and $B\,a_2$. The result is unclarity about how to check cross-type equality.

Reed [2003] and Abel and Scherer [2012] present similar type systems with irrelevant quantifiers in which this problem does not arise as they require the codomain $B : \mathsf{irr}\ A \to \mathcal{U}$ to be irrelevant as well. We can conclude that each of the concepts mentioned above has its own virtues. Irrelevance admits erasure at type-checking time, but we cannot consider irrelevant functions for an arbitrary dependent codomain. Erasability does allow arbitrary codomains, but admits erasure only at compile time. Parametricity does not admit erasure whatsoever, but it does admit pattern matching eliminators while still producing free theorems.

Cubical type theory and HoTT. Cubical type theory [Bezem et al. 2014; Cohen et al. 2016] uses a cubical set model, similar to ours, to model the univalence axiom from homotopy type theory (HoTT) and consequently, function extensionality. The cubical type system and ParamDTT have in common that equalities can be expressed using functions from the interval, and that types varying over the interval can be constructed using variations of Glue. We expect that both systems can be merged into a system for parametric HoTT. Voevodsky's homotopy type system [HTS, 2013] and Altenkirch et al.'s 2-level type system [2016], which contain both a fibrant path type and a non-fibrant strict equality type, may play well with such a system for parametric HoTT, where types live in a different context as their terms and hence need not be fibrant in their terms' context.

Iterated bridges. It is regrettable that we have lost internal iterated parametricity. This issue is directly related to the fact that in ParamDTT there is no way to provide, between two types A_0 and A_1, a notion of heterogeneous bridges without also providing a notion of heterogeneous paths. Indeed, if we have a bridge $A : \mathbb{I} \to \mathcal{U}$ from A_0 to A_1, then we can consider both bridges $\Pi(i : \mathbb{I}).A\,i$ and paths $\forall(i : \mathbb{I}).A\,i$. However, if we have a bridge between functions f and g, then a heterogeneous bridge from $a : \mathsf{Glue}\{A \leftarrow (P\,?\,T, f)\}$ to $b : \mathsf{Glue}\{A \leftarrow (P\,?\,T, g)\}$ has meaning in the model, whereas a path does not. This suggests that we should add to our model a weaker connection of *pro-bridges*, such that a pro-bridge between types expresses a notion of bridges, but not paths. This will then immediately ask for the addition of pro-pro-bridges, etc. It seems that a system for iterated bridge/path parametricity needs to be modelled in iterated bridge/path cubical sets which contain ever weaker notions of edges. On the syntax side, the consequence would be that the ¶ modality is lossless and we can have $\sharp \circ ¶ = \mathbf{id}$, which would stop the propagation of the ¶ modality that precludes iterated parametricity. A possibly related feature that ParamDTT lost with respect to both cubical type theory and Bernardy et al.'s work, is that our bridge and path types are not indexed by their endpoints; rather, they look like ordinary function spaces. The reason is that the interaction between modalities and indexed function types poses very subtle problems and we were able to achieve good results without. We believe that iterated bridge/path cubical sets could create clarity on this issue as well.

7 ACKNOWLEDGEMENTS

We want to thank Andreas Abel, Paolo Capriotti, Jesper Cockx, Dan Licata and Sandro Stucki for many fruitful discussions, and Andreas Abel, Sandro Stucki, Philip Wadler and the anonymous reviewers for their valuable feedback on the paper. Andreas Nuyts and Dominique Devriese hold a Ph.D. Fellowship and a Postdoctoral Mandate (resp.) from the Research Foundation - Flanders (FWO).

REFERENCES

Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. 1993. Formal parametric polymorphism. *Theoretical Computer Science* 121, 1 (1993), 9 – 58. DOI: http://dx.doi.org/10.1016/0304-3975(93)90082-5

Andreas Abel. 2006. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* Ph.D. Dissertation. Ludwig-Maximilians-Universität München.

Andreas Abel. 2008. Polarised subtyping for sized types. *Mathematical Structures in Computer Science* 18, 5 (2008), 797–822. DOI: http://dx.doi.org/10.1017/S0960129508006853

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012), 1–36. DOI: http://dx.doi.org/10.2168/LMCS-8(1:29)2012 TYPES'10 special issue.

Andreas M. Abel and Brigitte Pientka. 2013. Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 185–196. DOI: http://dx.doi.org/10.1145/2500365.2500591

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with Strict Equality. *CoRR* abs/1604.03799 (2016). http://arxiv.org/abs/1604.03799

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2006. Indexed Containers. Manuscript, available online. (February 2006).

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 16. 46–61. DOI: http://dx.doi.org/10.4230/LIPIcs.CSL.2012.46

Robert Atkey, Neil Ghani, and Patricia Johann. 2014. A Relationally Parametric Model of Dependent Type Theory. In *Principles of Programming Languages*. DOI: http://dx.doi.org/10.1145/2535838.2535852

Bruno Barras and Bruno Bernardo. 2008. *The Implicit Calculus of Constructions as a Programming Language with Dependent Types.* Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379. DOI: http://dx.doi.org/10.1007/978-3-540-78499-9_26

Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. *Electron. Notes in Theor. Comput. Sci.* 319 (2015), 67 – 82. DOI: http://dx.doi.org/10.1016/j.entcs.2015.12.006

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for Free — Parametricity for Dependent Types. *Journal of Functional Programming* 22, 02 (2012), 107–152. DOI: http://dx.doi.org/10.1017/S0956796812000056

Marc Bezem, Thierry Coquand, and Simon Huber. 2014. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 26. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 107–128. DOI: http://dx.doi.org/10.4230/LIPIcs.TYPES.2013.107

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a constructive interpretation of the univalence axiom. *CoRR* abs/1611.02108 (2016). http://arxiv.org/abs/1611.02108

Peter Dybjer. 1996. *Internal type theory.* Springer Berlin Heidelberg, Berlin, Heidelberg, 120–134. DOI: http://dx.doi.org/10.1007/3-540-61780-9_66

Ryu Hasegawa. 1994. Relational limits in general polymorphism. *Publications of the Research Institute for Mathematical Sciences* 30 (1994), 535–576.

Martin Hofmann. 1997. *Syntax and semantics of dependent types.* Springer London, London, Chapter 4, 13–54. DOI: http://dx.doi.org/10.1007/978-1-4471-0963-1_2

Martin Hofmann and Thomas Streicher. 1997. Lifting Grothendieck Universes. Unpublished note. (1997).

Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Computer Science Logic 2013 (CSL 2013) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 432–451. DOI: http://dx.doi.org/10.4230/LIPIcs.CSL.2013.432

Daniel Leivant. 1991. Finitely stratified polymorphism. *Information and Computation* 93, 1 (1991), 93 – 113. DOI: http://dx.doi.org/10.1016/0890-5401(91)90053-5

Daniel R. Licata and Robert Harper. 2011. 2-Dimensional Directed Type Theory. *Electronic Notes in Theoretical Computer Science* 276 (2011), 263 – 289. DOI: http://dx.doi.org/10.1016/j.entcs.2011.09.026 27th Conference on the Mathematical

Foundations of Programming Semantics.

Daniel R. Licata and Michael Shulman. 2016. *Adjoint Logic with a 2-Category of Modes.* Springer International Publishing, 219–235. DOI: http://dx.doi.org/10.1007/978-3-319-27683-0_16

Alexandre Miquel. 2001a. The Implicit Calculus of Constructions: Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications (TLCA'01).* Springer-Verlag, Berlin, Heidelberg, 344–359. http://dl.acm.org/citation.cfm?id=1754621.1754650

Alexandre Miquel. 2001b. *Le Calcul des Constructions Implicite: Syntaxe et Sémantique.* Ph.D. Dissertation. Université Paris 7.

Nathan Mishra-Linger and Tim Sheard. 2008. *Erasure and Polymorphism in Pure Type Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 350–364. DOI: http://dx.doi.org/10.1007/978-3-540-78499-9_25

Andreas Nuyts. 2017. *A Model of Parametric Dependent Type Theory in Bridge/Path Cubical Sets.* Technical Report. KU Leuven, Belgium. https://arxiv.org/abs/1706.04383

Frank Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science.* 221–230. DOI: http://dx.doi.org/10.1109/LICS.2001.932499

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. DOI: http://dx.doi.org/10.1017/S0960129501003322

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Gordon Plotkin and Martín Abadi. 1993. *A logic for parametric polymorphism.* Springer Berlin Heidelberg, Berlin, Heidelberg, 361–375. DOI: http://dx.doi.org/10.1007/BFb0037118

Jason Reed. 2003. Extending Higher-Order Unification to Support Proof Irrelevance. In *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003, Rome, Italy, September 8-12, 2003. Proceedings*, David Basin and Burkhart Wolff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–252. DOI: http://dx.doi.org/10.1007/10930755_16

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism.. In *IFIP Congress.* 513–523.

John C. Reynolds. 1984. *Polymorphism is not set-theoretic.* Research Report RR-0296. INRIA. https://hal.inria.fr/inria-00076261

Edmund P. Robinson and Giuseppe Rosolini. 1994. Reflexive graphs and parametric polymorphism. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science.* 364–371. DOI: http://dx.doi.org/10.1109/LICS.1994.316053

Izumi Takeuti. 2001. *The Theory of Parametricity in Lambda Cube.* Technical Report 1217. Kyoto University.

Vladimir Voevodsky. 2013. A simple type system with two identity types. (2013). https://ncatlab.org/homotopytypetheory/files/HTS.pdf unpublished note.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89).* ACM, New York, NY, USA, 347–359. DOI: http://dx.doi.org/10.1145/99370.99404

Philip Wadler. 2007. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science* 375, 1 (2007), 201 – 226. DOI: http://dx.doi.org/10.1016/j.tcs.2006.12.042

Chapter 6

Normalization by evaluation for sized dependent types

Normalization by Evaluation for Sized Dependent Types

ANDREAS ABEL, Gothenburg University, Sweden
ANDREAS VEZZOSI, Chalmers University of Technology, Sweden
THEO WINTERHALTER, École normale supérieure Paris-Saclay, France

Sized types have been developed to make termination checking more perspicuous, more powerful, and more modular by integrating termination into type checking. In dependently-typed proof assistants where proofs by induction are just recursive functional programs, the termination checker is an integral component of the trusted core, as validity of proofs depend on termination. However, a rigorous integration of full-fledged sized types into dependent type theory is lacking so far. Such an integration is non-trivial, as explicit sizes in proof terms might get in the way of equality checking, making terms appear distinct that should have the same semantics.

In this article, we integrate dependent types and sized types with higher-rank size polymorphism, which is essential for generic programming and abstraction. We introduce a size quantifier $\forall$ which lets us ignore sizes in terms for equality checking, alongside with a second quantifier Π for abstracting over sizes that do affect the semantics of types and terms. Judgmental equality is decided by an adaptation of normalization-by-evaluation for our new type theory, which features *type shape*-directed reflection and reification. It follows that subtyping and type checking of normal forms are decidable as well, the latter by a bidirectional algorithm.

CCS Concepts: • **Theory of computation → Type theory**; **Type structures**; *Program verification*; Operational semantics;

Additional Key Words and Phrases: dependent types, eta-equality, normalization-by-evaluation, proof irrelevance, sized types, subtyping, universes.

ACM Reference format:
Andreas Abel, Andreas Vezzosi, and Theo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 33 (September 2017), 30 pages.
https://doi.org/10.1145/3110277

1 INTRODUCTION

Dependently-typed programming languages and proof assistants, such as Agda [2017] and Coq [INRIA 2016], require programs to be total, for two reasons. First, for consistency: since propositions are just types and proofs of a proposition just programs which inhabit the corresponding type, some types need to be empty; otherwise, each proposition would be true. However, in a partial language with general recursion, each type is inhabited by the looping program $f = f$. Secondly, totality is needed for decidability of type checking. Since types can be the result of a computation, we need computation to terminate during type checking, even for *open terms*, i. e., terms with free variables.

Consequently, the aforementioned languages based on Type Theory come with a termination checker, which needs to reject all non-terminating programs, and should accept sufficiently many

2475-1421/2017/9-ART33
https://doi.org/10.1145/3110277

terminating programs to allow the user to express her algorithms. In current termination checkers, programs are required to terminate by structural descent [Giménez 1995]; the structural order may be extended to a lexicographic [Abel and Altenkirch 2002] or size-change termination criterion [Lee et al. 2001; Wahlstedt 2007]. This is not a fundamental limitation, since Type Theory allows many functions to be expressed in a structurally recursive manner, if needed by the help of a well-founded relation [Nordström 1988], inductive domain predicates [Bove and Capretta 2005], or inductive descriptions of the function graph [Bove 2009]. However, the syntactic termination check is very sensitive to reformulations of the program and hostile to abstraction [Abel 2012].

Sized types [Hughes et al. 1996] delegate the checking for structural descent to the type system by annotating data types with a size parameter. The type checker can then ensure that in recursive calls the size goes down, certifying termination. In the simplest setting [Abel 2008; Barthe et al. 2004], the size is just an upper bound on the tree height of the data structure; however, more sophisticated size annotations have also been considered [Blanqui 2004; Xi 2002]. Most sized type systems are non-dependent [Abel and Pientka 2016; Amadio and Coupet-Grimal 1998; Barthe et al. 2008a,b; Blanqui and Riba 2006; Lago and Grellois 2017], yet the combination of sized and dependent types has been studied as well [Barthe et al. 2006; Blanqui 2005; Grégoire and Sacchini 2010; Sacchini 2013, 2014]. However, to the best of our knowledge, no study combines *higher-rank size polymorphism* with full-fledged dependent types.[1]

Higher-rank size quantification takes termination checking to the next level; it is necessary for abstraction and generic programming. For instance, it allows us to write a generic tree traversal which applies a user-given preprocessor on subtrees before recursively descending into these trees, and a postprocessor after surfacing from the descent. The condition is that preprocessing does not increase the size of the subtree; otherwise, termination could not be guaranteed. Concretely, assume a type $\mathsf{T}\,i$ of trees of size $< i$ with a constructor $\mathsf{node} : \forall i.\, \mathsf{List}\,(\mathsf{T}\,i) \to \mathsf{T}\,(i+1)$ which takes a finite list of subtrees to form a new tree. In the following definition of trav, the preprocessing $\mathit{pre} : \forall i.\, \mathsf{T}\,i \to \mathsf{T}\,i$ can be safely applied to input tree t because the type of pre bounds the size of $\mathit{pre}\,t$ by the size of t. In case $\mathit{pre}\,t = \mathsf{node}\,\mathit{ts}$, the trees in the list ts are still guaranteed to be of strictly smaller size than t, thus, the recursive call to trav, communicated via the map function for lists, is safe.

$$\begin{aligned}&\mathsf{trav} : (\mathit{pre} : \forall i.\, \mathsf{T}\,i \to \mathsf{T}\,i)\,(\mathit{post} : \mathsf{T}\infty \to \mathsf{T}\infty) \to \forall i.\, \mathsf{T}\,i \to \mathsf{T}\,\infty\\ &\mathsf{trav}\ \mathit{pre}\ \mathit{post}\ t = \mathit{post}\ (\mathsf{case}\ \mathit{pre}\,t\ \mathsf{of}\ \{\,\mathsf{node}\ \mathit{ts} \to \mathsf{node}\,(\mathsf{map}\ (\mathsf{trav}\ \mathit{pre}\ \mathit{post})\ \mathit{ts})\,\})\end{aligned}$$

The display shows the *Curry*-style program as provided by the user, but state-of-the-art type checkers elaborate the program from surface syntax into an internal *Church*-style syntax with explicit type abstractions and type applications.[2] With implicit type and size applications elaborated, trav would look as follows:

$$\begin{aligned}&\mathsf{trav}\ \mathit{pre}\ \mathit{post}\ i\ t =\\ &\quad \mathit{post}\ (\mathsf{case}\ \mathit{pre}\,i\,t\ \mathsf{of}\ \{\,\mathsf{node}\,j\,\mathit{ts} \to \mathsf{node}\,\infty\,(\mathsf{map}\ (\mathsf{T}\,j)\ (\mathsf{T}\,\infty)\ (\mathsf{trav}\ \mathit{pre}\ \mathit{post}\,j)\ \mathit{ts})\})\end{aligned}$$

Church-style syntax is the basis for all program analyses and transformations to follow and should be considered as the *true* syntax. However, from a dependent-type perspective, explicit size applications in terms can be problematic when the type checker compares terms for equality—which is necessary as types can depend on values. Inferred sizes may not be unique, as we have subtyping $\mathsf{T}\,i \le \mathsf{T}\,j$ for $i \le j$: we can always weaken an upper bound. For instance, given $\mathit{ts} : \mathsf{List}\,(\mathsf{T}\,i)$, any of the terms $\mathsf{node}\,i\,\mathit{ts}$, $\mathsf{node}\,(i+1)\,\mathit{ts}$, …, $\mathsf{node}\,\infty\,\mathit{ts}$ has type $\mathsf{T}\,\infty$. Yet semantically, all these trees are equal, thus, the syntactic equality check should ignore the size argument to node. Similarly, in the

[1] Xi [2002] has first-class size polymorphism, but only indexed types, no universes or large eliminations.
[2] Agda, Coq, Idris [Brady 2013], and Haskell [Sulzmann et al. 2007] all have Church-style internal languages.

application $\mathit{pre}\, i\, t$ the size argument i should be ignored by the equality check. Yet $\mathit{pre}\, i : \mathsf{T}\, i \to \mathsf{T}\, i$ and $\mathit{pre}\, j : \mathsf{T}\, j \to \mathsf{T}\, j$ have different types for $i \neq j$, and moreover these function types are not in the subtyping relation due to the mixed-variant occurrence of the size parameter. It seems that during equality checking we have to consider terms of different types, at least for a while. Once we apply $\mathit{pre}\, i$ and $\mathit{pre}\, j$ to the same tree $t : \mathsf{T}\, k$, which determines $i = j = k$, we are back to safety. However, allowing types to differ during an equality check needs special consideration, especially when the equality-check is type directed.

Consider the analogous situation for the polymorphic lambda calculus System F, be it the predicative variant or not, extended by a unit type $\mathbf{1}$. For Church-style, we can give a type-directed $\beta\eta$-equality test which equates all terms at the unit type. The most interesting rules are the η-rules for unit and function type and the congruence rule for type application:

$$\frac{}{\Gamma \vdash t = t' : \mathbf{1}} \qquad \frac{\Gamma, x{:}A \vdash t\, x = t'\, x : B}{\Gamma \vdash t = t' : A \to B} \qquad \frac{\Gamma \vdash t = t' : \forall X.B}{\Gamma \vdash t\, A = t'\, A : B[A/X]}$$

The Curry-style version replaces the last conclusion by $\Gamma \vdash t = t' : B[A/X]$ where the type A to instantiate X has to be guessed. However, in Curry-style more terms are equated than in Church-style, as for instance the Church-style terms $t\, A\, (\lambda x : A.\, x)$ and $t\, B\, (\lambda x : B.\, x)$ map to the same Curry-style term $t\, (\lambda x.\, x)$. How would we adapt the algorithm for Church-style such that it equates all terms that are equal in Curry-style? The conclusion of the last rule could be changed to $\Gamma \vdash t\, A = t'\, A' : B[A/X]$, but then the second term $t'\, A'$ does not have the ascribed type $B[A/X]$, and η-laws applied to this term would be unsound. For instance, the algorithm would yield $t\, \mathbf{1}\, x = t\, (A \to A)\, y$ even for $x \neq y$. We could also consider a heterogeneous check $(\Gamma \vdash t : A) = (\Gamma' \vdash t' : A')$ where each term is paired with its own type and context, but this leaves us with the dilemma of explaining the meaning of this judgement when A and A' are incompatible.

Does the literature offer a solution to this problem? In fact, a Church-style calculus with Curry-style equality has been studied before, it is ICC* [Barras and Bernardo 2008; Mishra-Linger and Sheard 2008] based on Miquel's Implicit Calculus of Constructions [2001]. In ICC*, equality is checked by erasing all type abstractions and applications, and comparing the remaining untyped terms for $\beta\eta$-equality. While this works for η-laws that can be formulated on untyped terms, such as η-contraction of functions $\lambda x.\, t\, x \longrightarrow_\eta t$ (when x not free in t), it does not extend to type-directed η-laws such as extensionality for the unit type. Further, ICC* is not a type theory formulated with a typed equality judgement, which makes it hard to define its models [Miquel 2000]—we wish not to go there, but stay within the framework of Martin-Löf Type Theory [1975].

Now, if the types of compared Curry-style terms are not equal, can they be sufficiently related to give a proper meaning to the algorithmic equality judgement? It has already been observed that for a type-directed equality check the precise type is not necessary, a *shape* or *skeleton* is sufficient. The skeleton informs the algorithm whether the terms under comparison are functions, inhabitants of the unit type, or something else, to possibly apply the appropriate η-law. For the Logical Framework (LF), the simplest dependent lambda-calculus, the skeletons are simple types that can be obtained from the original dependent types by erasing the dependencies: dependent function types map to non-dependent ones and indexed data types to simple data types. Harper and Pfenning [2005] present such an equality check for LF which is directed by simple types, and their technique should scale to other type theories that admit dependency erasure.[3]

[3] For instance, the types of the Calculus of Constructions erase to F^ω-types [Geuvers 1994], and the latter could be used to direct the equality check. Lovas and Pfenning [2010] consider also *refinement types for logical frameworks* which can be erased to simple types.

By *large eliminations* [Werner 1992] we refer to types computed by case distinction over values; they occur in type theories that feature both universes and data types. In the presence of large eliminations, dependency erasure fails, and it is not clear what the skeleton of a dependent type should be. For instance consider the type $(n{:}\mathbb{N}) \to \underbrace{A \to \cdots \to A}_{n} \to A$ of n-ary functions; its shape is dependent on the value of n, thus cannot be determined statically. Thus, the "skeleton" idea is also not directly applicable.

Going beyond the standard syntax-directed equality check, there is a technique that can deal with dynamic η-expansion. It is a type-directed normalization function inspired by normalization-by-evaluation (NbE) that computes η-long normal forms [Berger and Schwichtenberg 1991; Danvy 1999]. We can check the computed normal forms for identity and, thus, decide definitional equality. NbE has proven to be a robust method to decide equality in powerful type theories with non-trivial η-laws. It scales to universes and large eliminations [Abel et al. 2007], topped with singleton types or proof irrelevance [Abel et al. 2011], and even impredicativity [Abel 2010]. At its heart there are reflection $\uparrow^T$ and reification $\downarrow^T$ functions directed by type T and orchestrating just-in-time η-expansion. Reflection $\uparrow^T x$ maps variables x into the realm of values of type T and lets us compute with open terms. Reification $\downarrow^T a$ takes a value a of type T and computes its long normal form. For instance, the normal form of a closed function $f : U \to T$ would be $\lambda x.\downarrow^T (f\,(\uparrow^U x))$, and for its dependently-typed variant $f : (x{:}U) \to T[x]$ it would be $\lambda x.\downarrow^{T[\uparrow^U x]} (f\,(\uparrow^U x))$.

The central technical observation is that reflection and reification do not need the precise type T, they work the same for any *shape* S of T. We managed, while not introducing a new syntax for shapes, to define a relation $T \sqsubseteq S$ on type values stating that type S qualifies as shape for type T. Shapes unfold dynamically during reflection and reification. For example, when reflecting a variable x into the polymorphic function type $\forall i.\, F\,i$ where $F\,i = \mathsf{Nat}\ i \to \mathsf{Nat}\ i$, we obtain $(\uparrow^{\forall i.F\,i} x)\,i = \uparrow^{F\,i}(x\,i)$ for size i and $(\uparrow^{\forall i.F\,i} x)\,j = \uparrow^{F\,j}(x\,j)$ for size j. The new types $F\,i$ and $F\,j$ we reflect at are no longer equal (and they are not subtypes of each other), but they still have the same shape, $\mathsf{Nat}\ \_ \to \mathsf{Nat}\ \_$. This means they will still move in lock-step in respect to η-expansion, which is sufficient to prove NbE correct for judgmental equality. We call the enabling property of F *shape irrelevance*, meaning that for any pair i, j of legal arguments, $F\,i$ and $F\,j$ have the same shape. Whenever we form a irrelevant function type $\forall x{:}U.\, T[x]$, we require $T[x]$ to be shape-irrelevant in x. This is the middle ground between ICC*, where no restriction is placed on T but η for unit types is out of reach (at least for the moment), and Pfenning's [2001] irrelevance modality, adapted to full dependent types by Abel and Scherer [2012], which requires T to be irrelevant in x and, thus, has type equality $T[i] = T[j]$.

For the time being, we do not (and cannot) develop a general theory of shape irrelevance. We confine ourselves to size-irrelevant function types $\forall i.\, T[i]$. This relieves us from defining a special shape-irrelevance modality, since all size-indexed types $T[i]$ *are* shape irrelevant in i, simply because there is no case distinction on size, and sizes appear relevantly only under a sized type constructor such as Nat. Our technique would not extend to the polymorphic types $\forall X.\, B[X]$ of System F. Even though there is no case distinction on types, shape irrelevance of $B[X]$ fails in general, as X could appear as a type on the top-level, e.g. in $B[X] = X \to X$, and then $B[\mathbf{1}]$ and $B[A \to A]$ would have distinct shapes.

To summarize, this article makes the following novel contributions:

(1) We present the first integration of a dependent type theory with higher-rank size polymorphism. Concretely, we consider a type theory à la Martin-Löf with dependent function types, cumulative universes, subtyping, a judgmental equality with η-laws, a sized type of natural numbers and two size quantifiers: an irrelevant one ($\forall$) for binding of sizes in irrelevant

positions, and a relevant one (Π) for binding of sizes in shape-irrelevant positions (Section 3). Judgmental equality features a "Curry-style" rule for irrelevant size application which ignores the size arguments, and consequently, the corresponding typing rule will also ignore the size argument. (In the following rules, a, a', and b stand for arbitrary size expressions.)

$$\frac{\Gamma \vdash t = t' : \forall i.\, T}{\Gamma \vdash t\, a = t'\, a' : T[b/i]} \qquad \frac{\Gamma \vdash t : \forall i.\, T}{\Gamma \vdash t\, a : T[b/i]} \qquad \frac{\Gamma \vdash t = t' : \Pi i.\, T}{\Gamma \vdash t\, a = t'\, a : T[a/i]} \qquad \frac{\Gamma \vdash t : \Pi i.\, T}{\Gamma \vdash t\, a : T[a/i]}$$

Our substitution theorem distinguishes term- from type-side substitutions.

(2) We adapt normalization-by-evaluation (NbE) to sized types and size quantification and show that it decides judgmental equality (sections 4 and 5). The novel technical tool is a relation $T \sqsubseteq S$ which relates a type T to its possible shapes S. This approximation relation allows reflection and reification at size-polymorphic types $\forall i.\, T$. As usual for the meta-theory of Type Theory with large eliminations, the machinery is involved, but we just require the usual two logical relations: First, a PER model to define the semantics of types and prove the completeness of NbE (Section 4). Secondly, a relation between syntax and semantics to prove soundness of NbE (Section 5).

(3) We present an bidirectional type checking algorithm [Coquand 1996] which takes the irrelevant size argument as reliable hint for the type checker (sections 6 and 7). It is complete for normal forms which can be typed with the restricted rule for ∀-elimination:

$$\frac{\Gamma \vdash t : \forall i.\, T}{\Gamma \vdash t\, a : T[a/i]}$$

The algorithm employs the usual lazy reduction for types, i. e., just-in-time weak-head evaluation, in type and sub-type checker [Huet 1989]. In this, it improves on Fridlender and Pagano [2013] which instruments full normalization (NbE) at every step.

This article is accompanied by a prototypical type checker `Sit` which implements the type system and type checking algorithm as described in the remainder of the paper. But before going into the technical details, we will motivate our type system from a practical perspective: reasoning about programs involving sized types in Agda.

2 SIZE IRRELEVANCE IN PRACTICE

In this section, we show how the lack of size irrelevance prevents us from reasoning naturally about programs involving sized types in Type Theory. We focus on Agda, at the time of writing the only mature implementation of Type Theory with an experimental integration of sized types.

The problem of the current implementation of sized types in Agda can be demonstrated by a short example. Consider the type of sized natural numbers.

```
data Nat : Size → Set where
  zero : ∀ i → Nat (i + 1)
  suc  : ∀ i → Nat i → Nat (i + 1)
```

The predecessor function is *size preserving*, i. e., the output can be assigned the same upper bound i as the input. In the code to follow, the dot on the left hand side, preceding $(i + 1)$, marks an *inaccessible* pattern. Its value is determined by the subsequent match on the natural number argument, no actual matching has to be carried out on this argument.

```
pred : ∀ i → Nat i → Nat i
pred .(i + 1) (zero i)   = zero i
pred .(i + 1) (suc i x) = x
```

Note that in the second clause, we have applied subtyping to cast x from Nat i to Nat $(i + 1)$.

We now define subtraction $x \dot{-} y$ on natural numbers, sometimes called the monus function, which computes $\max(0, x - y)$. It is defined by induction on the size j of the second argument y, while the output is bounded by size i of the first argument x. The input-output relation of monus is needed for a natural implementation of Euclidean divison.

There are several ways to implement monus, we have chosen a tail-recursive variant which treats the first argument as accumulator. It computes the result by applying the predecessor function y times to x.

```
monus : ∀ i → Nat i → ∀ j → Nat j → Nat i
monus i x .(j + 1) (zero j)   = x
monus i x .(j + 1) (suc j y) = monus i (pred i x) j y
```

To document subgoals in proof terms, we introduce a mixfix version of the identity function with a visible type argument:

```
prove_by_ : (A : Set) → A → A
prove A by x = x
```

We now wish to prove that subtracting x from itself yields 0, by induction on x. The case $x = 0$ should be trivial, as $x \dot{-} 0 = x$ by definition, hence, $0 \dot{-} 0 = 0$. As simple proof by reflexivity should suffice. In case $x + 1$, the goal $0 \equiv (x + 1) \dot{-} (x + 1)$ should reduce to $0 \equiv x \dot{-} x$, thus, an application of the induction hypothesis should suffice. The following display shows that partial proofs, leaving holes {! ... !} already filled with the desired proof terms.

```
monus-diag : ∀ i → (x : Nat i) → zero ∞ ≡ monus i x i x
monus-diag .(i + 1) (zero i)   = prove zero ∞ ≡ zero i                by {! refl !}
monus-diag .(i + 1) (suc i x) = prove zero ∞ ≡ monus (i + 1) x i x by {! monus-diag i x !}
```

Unfortunately, in Agda our proof is not accepted, as sizes get in the way. In the first goal, there is a mismatch between size ∞ and size i, the latter coming from the computation of monus $(i + 1)$ (zero i) $(i+1)$ (zero i). In the second goal, there is a mismatch between size $i+1$ in term monus $(i+1)\, x\, i\, x$ of the reduced goal and size i of the respective term monus $i\, x\, i\, x$ from the induction hypothesis we wish to apply.

The proof would go through if Agda ignored sizes where they act as *type argument*, i. e., in constructors and term-level function applications, but not in types where they act as regular argument, e. g., in Nat i.

The solution we present in this article already works in current Agda,[4] but the implementation is not perfect. Thus, it is hidden under a scarcely documented flag:

```
{-# OPTIONS --experimental-irrelevance #-}
```

We mark the size argument of Nat as *shape irrelevant* by preceding the binder with two dots. In a future implementation, we could treat all data type parameters as shape irrelevant by default. In the types of the constructors, we mark argument i as *irrelevant* by prefixing the binder with a single dot. This is sound because i occurs in subsequent parts of the type only in shape-irrelevant positions.

```
data Nat : ..(i : Size) → Set where
  zero : ∀ .i → Nat (i + 1)
  suc  : ∀ .i → Nat i → Nat (i + 1)
```

[4] https://github.com/agda/agda, development version of 2017-02-27.

Similarly, "type" argument i to pred is irrelevant. Agda checks that it only occurs shape-irrelevantly in the type and irrelevantly in the term. The latter is the case since i is also an irrelevant argument to the constructors zero and suc; otherwise, we would get a type error.

```
pred : ∀ .i → Nat i → Nat i
pred .(i + 1) (zero i)  = zero i
pred .(i + 1) (suc i x) = x
```

The two size arguments i and j to monus are also irrelevant. In this case, type checking succeeds since the size argument to pred has been declared irrelevant.

```
monus : ∀ .i → Nat i → ∀ .j → Nat j → Nat i
monus i x .(j + 1) (zero j)  = x
monus i x .(j + 1) (suc j y) = monus i (pred i x) j y
```

Now, with sizes being ignored in the involved terms, we can complete the proof of our lemma:

```
monus-diag : ∀ .i → (x : Nat i) → zero ∞ ≡ monus i x i x
monus-diag .(i + 1) (zero i)  = prove zero ∞ ≡ zero i                by refl
monus-diag .(i + 1) (suc i x) = prove zero ∞ ≡ monus (i + 1) x i x by monus-diag i x
```

3 A TYPE SYSTEM WITH IRRELEVANT SIZE APPLICATION

In this section, we give the syntax and the *declarative* typing, equality, and subtyping judgements. The typing relation $\Gamma \vdash t : T$ will *not* be decidable; instead, we present algorithmic typing $\Gamma \vdash t \leftleftarrows T$ in Section 7. However, equality and subtyping will be decidable for well-formed input, see sections 4–6.

We present our type theory as (domain-free) *pure type system* [Barendregt 1991] with extra structure. The *sorts* s are drawn from an infinite predicative hierarchy of universes Set_ℓ for $\ell \in \mathbb{N}$. Universes provide us with polymorphism and the capability to define types by recursion on values. Whether we have just two universes Set_0 and Set_1 or infinitely many, does not matter for the technical difficulty of the meta theory. The present setup have the advantage that every sort has again a sort since $\mathrm{Set}_\ell : \mathrm{Set}_{\ell+1}$, thus, we do not have to introduce a separate judgement $\Gamma \vdash T$ for well-formedness of types, we can define it as $\exists s.\ \Gamma \vdash T : s$.

Sort	$\ni s$	$::=$	$\mathrm{Set}_\ell\ (\ell \in \mathbb{N})$	sort (universe)
Ann	$\ni \star$	$::=$	$\div \mid :$	annotation (irrelevant, relevant)
Exp	$\ni t, u, T, U$	$::=$	$w \mid t\ e$	expressions
Whnf	$\ni w, W$	$::=$	$n \mid s \mid \mathrm{Size} \mid \Pi^\star U\ T \mid \lambda t \mid \mathrm{Nat}\ a \mid c$	weak head normal forms
Data	$\ni c$	$::=$	$\mathrm{zero}\langle a\rangle \mid \mathrm{suc}\langle a\rangle t$	constructed data
NeExp	$\ni n$	$::=$	$\mathsf{v}_i \mid n\ e$	neutral expressions
Elim	$\ni e$	$::=$	$t \mid a \mid \langle a\rangle \mid \mathrm{case}_\ell\ T\ t_z\ t_s \mid \mathrm{fix}_\ell\ T\ t$	eliminations
SizeExp	$\ni a, b$	$::=$	$\infty \mid o \mid \mathsf{v}_i + o\ (o \in \mathbb{N})$	size expressions
Cxt	$\ni \Gamma, \Delta$	$::=$	$() \mid \Gamma.^\cdot T \mid \Gamma.^\div \mathrm{Size}$	contexts
Subst	$\ni \eta, \rho, \sigma, \tau, \xi$	$::=$	$() \mid (\sigma, t)$	substitutions

Fig. 1. Syntax.

For the expression syntax (see Fig. 1), we use de Bruijn [1972] indices v_i to represent variables. The index $i \in \mathbb{N}$ points to the ith enclosing binder of variable v_i. Binders are lambda abstraction λt and

dependent function types $\Pi^\star U\,T$, which bind the 0th index in t and T, resp. For instance, the term $\lambda x.\,x\,(\lambda z.\,z)\,(\lambda y.\,x\,y)$ with named variables x, y, z has de Bruijn representation $\lambda.\,\mathsf{v}_0\,(\lambda.\,\mathsf{v}_0)\,(\lambda.\,\mathsf{v}_1\,\mathsf{v}_0)$.

The notation $\Pi^\star U\,T$ is an umbrella for three kinds of function types, where $\star \in \{\div, :\}$ is a relevance annotation borrowed from Pfenning [2001]. $\Pi^{:} U\,T$ is the ordinary dependent function type, $\Pi^{:}\mathsf{Size}\,T$ is relevant size quantification, and $\Pi^{\div}\mathsf{Size}\,T$ is irrelevant size quantification. We omit the ":"-markers from Π by default (and also in contexts Γ) and write $\forall\,T$ for $\Pi^{\div}\mathsf{Size}\,T$. Examples for relevant size quantification $\Pi\,\mathsf{Size}\,T$ are $\Pi\,\mathsf{Size}\,\mathsf{Set}_0$ and $\Pi\,\mathsf{Size}\,\Pi\,(\mathsf{Nat}\,\mathsf{v}_0)\,\mathsf{Set}_0$. In a syntax with named variables and non-dependent function type they could be written as $\mathsf{Size} \to \mathsf{Set}_0$ and $(z : \mathsf{Size}) \to \mathsf{Nat}\,z \to \mathsf{Set}_0$, resp. An instance of irrelevant quantification $\forall T$ would be $\forall.\,\Pi\,(\mathsf{Nat}\,\mathsf{v}_0)\,(\mathsf{Nat}\,\mathsf{v}_1)$ which is $\forall z.\,\mathsf{Nat}\,z \to \mathsf{Nat}\,z$ in a named syntax. Herein, $\mathsf{Nat}\,z$ denotes the type of natural numbers below z. The expression Size is a possible instance of U in $\Pi^\star U\,T$, or a possible type of a variable in a typing context Γ, but not a first-class type, i. e., we cannot construct our own functions on sizes.

Canonical natural numbers c are constructed by $\mathsf{zero}\langle a\rangle$ and $\mathsf{suc}\langle a\rangle t$. A size expression a is either a constant $o \in \mathbb{N}$, a variable $\mathsf{v}_i + o$ possibly with increment o, or the limit ordinal ∞ which stands for ω. The size argument a in the constructors zero and suc is a suggestion for the type checker but bears no semantic significance. For example, in the declarative typing presented here, we can have $\vdash \mathsf{zero}\langle 5\rangle : \mathsf{Nat}\,1$. In the algorithmic typing however, $\vdash \mathsf{zero}\langle 5\rangle \mathrel{\reflectbox{$\in$}} \mathsf{Nat}\,1$ will be an error. Note, however, that $\vdash \mathsf{zero}\langle a\rangle : \mathsf{Nat}\,0$ is impossible for any a, as zero is not strictly below 0 (when both term and size are interpreted as natural numbers).

Regular application $t\,u$, relevant size application $t\,a$, and irrelevant size application $t\,\langle a\rangle$ eliminate functions t and are subsumed under the form $t\,e$ with $e ::= u \mid a \mid \langle a\rangle$. We have two further eliminations, which make sense when t stands for a natural number. These are case distinction $e = \mathsf{case}_\ell\,T\,t_z\,t_s$ and recursive function application $e' = \mathsf{fix}_{\ell'}\,T'\,t'$. Application of case distinction $\mathsf{zero}\langle a\rangle\,e$ will reduce to the zero-branch t_z, and application $(\mathsf{suc}\langle a\rangle t)\,e$ to the instantiation $t_s\,t$ of the successor branch. The type annotation T in case allows us to infer the type of the whole case statement $t\,e$ as $T\,t$. The function call $c\,e'$ for a canonical number c and elimination $e' = \mathsf{fix}_{\ell'}\,T'\,t'$ reduces to $t'\,(\lambda x.\,x\,e')\,c$ where we allowed ourselves the use of a named abstraction in the presentation to the reader. The unfolding of fixed-points is thus restricted to application to canonical numbers; this is the usual reduction strategy which converges for terminating functions [Barthe et al. 2004].

For ordinary β-reduction we employ *substitutions* σ. These are simply lists of terms that provide one term as replacement for each free de Bruijn index in a term t. We write $\boxed{t\sigma}$ for the application of substitution σ to term t which is defined as usual. Let *lifting* $\boxed{\uparrow^k_m}$ be the substitution $(\mathsf{v}_{k+m-1}, \ldots, \mathsf{v}_{k+1}, \mathsf{v}_k)$ which accepts a term with m free indices and increases each of them by k. We write $\uparrow_m$ for the lifting $\uparrow^1_m$ and $\boxed{\mathsf{id}_m}$ for the identity substitution $\uparrow^0_m$. In general, we refer to liftings by letter ξ. The substitution $\boxed{[u]_m} = (\mathsf{id}_m, u)$ replaces free index v_0 by term u and decrements the other m free indices by 1. We drop subscript m from liftings and substitutions when clear from the context. Substitution composition $\boxed{\sigma\tau}$ is the pointwise application of substitution τ to the list of terms σ. In the proofs to follow, we freely use the following identities:

$$t\,\mathsf{id} \equiv t \qquad (t\sigma)\tau \equiv t(\sigma\tau) \qquad \sigma\,\mathsf{id} \equiv \sigma \qquad \mathsf{id}\,\tau \equiv \tau \qquad (\rho\sigma)\tau \equiv \rho(\sigma\tau)$$
$$\mathsf{v}_0(\sigma, t) \equiv t \qquad \uparrow(\sigma, t) \equiv \sigma \qquad [t]\sigma \equiv (\sigma, t\sigma) \qquad \uparrow[t] \equiv \mathsf{id}$$

As already done in some examples, we may use a named dependent function type notation as syntactic sugar for the corresponding de Bruijn representation. For instance, $(z : \mathsf{Size}) \to \mathsf{Nat}\,z \to \mathsf{Set}_\ell$ is sugar for $\Pi\,\mathsf{Size}\,\Pi\,(\mathsf{Nat}\ \mathsf{v}_0)\,\mathsf{Set}_\ell$. We abbreviate this type by $\boxed{\mathsf{FixK}\,\ell}$, and let $\boxed{\mathsf{FixT}\,T}$ stand for $\forall z.\,((x : \mathsf{Nat}\,z) \to T\,z\,x) \to (x : \mathsf{Nat}\,(z+1)) \to T\,(z+1)\,x$. Similarly to for Π, we use named

lambda abstraction as sugar for de Bruijn abstraction. Named abstraction takes care of proper lifting of de Bruijn indices, for instance, $\lambda x.\, t x = \lambda.(t\uparrow)\, \mathsf{v}_0$ if t is outside the scope of x. We may also use names when we construct concrete contexts, for instance, if T is well-formed in context Γ, we may write $T\, z\, x$ in context $\Gamma.z{:}\mathsf{Size}.x{:}\mathsf{Nat}\, z$ to mean $T\uparrow^2 \mathsf{v}_1\, \mathsf{v}_0$ in context $\Gamma.\mathsf{Size}.\mathsf{Nat}\, \mathsf{v}_0$.

Inductively defined judgements (mutual).

$\vdash \Gamma$	context Γ is well-formed
$\Gamma(i) = {}^{\star}T$	in context Γ, de Bruijn index i has type T and annotation $\star$
$\Gamma \vdash a : \mathsf{Size}$	in context Γ, size expression a is well-formed
$\Gamma \vdash t : T$	in context Γ, term t has type T
$\Gamma \vdash t = t' : T$	in context Γ, terms t and t' are equal of type T
$\Gamma \vdash T \leq T'$	in context Γ, type T is a subtype of T'
$\Gamma \vdash \sigma : \Delta$	σ is a valid substitution for Δ
$\Gamma \vdash \sigma = \sigma' \coloneqq \tau : \Delta$	$\sigma/\sigma'/\tau$ are a equal term/term/type-level substitutions for Δ

Derived judgements.

$\Gamma \vdash T$	$:\Longleftrightarrow$	$\Gamma \vdash T : s$ for some s
$\Gamma \vdash T = T'$	$:\Longleftrightarrow$	$\Gamma \vdash T = T' : s$ for some s
$\Gamma \vdash a = b : \mathsf{Size}$	$:\Longleftrightarrow$	$\Gamma \vdash a : \mathsf{Size}$ and $a = b$
$\Gamma \vdash a \leq b : \mathsf{Size}$	$:\Longleftrightarrow$	$\Gamma \vdash a : \mathsf{Size}$ and $\Gamma \vdash b : \mathsf{Size}$ and $a \leq b$
$\Gamma \vdash T : \mathsf{Adm}\, \ell$	$:\Longleftrightarrow$	$\Gamma \vdash T : \mathsf{FixK}\, \ell$ and $\Gamma.z{:}\mathsf{Size}.x{:}\mathsf{Nat}\, z \vdash T\, z\, x \leq T\, \infty\, x$
$\xi : \Gamma \leq \Delta$	$:\Longleftrightarrow$	$\Gamma \vdash \xi : \Delta$ and $\xi = \uparrow^k_m$ with $m = \lvert\Delta\rvert$ and $k = \lvert\Gamma\rvert - m$

Fig. 2. Judgements.

In typing contexts Γ, we distinguish relevant ($:$) and irrelevant ($\div$) bindings. When type checking a variable, it needs to be bound in the context relevantly. However, when entering an irrelevant position, for instance when checking size a in term $\mathsf{suc}\langle a \rangle t$ we declare previously irrelevant variables as relevant. This operation on the context has been coined *resurrection* by Pfenning [2001]; formally $\boxed{\Gamma^{\oplus}}$ removes the "$\div$"-markers from all bindings in Γ, i. e., replaces them by "$:$"-markers. Note that, trivially, resurrection is idempotent: $\Gamma^{\oplus\oplus} = \Gamma^{\oplus}$.

Size increment $\boxed{a + o'}$ for $o' \in \mathbb{N}$ extends addition by $\infty + o' = \infty$ and $(\mathsf{v}_i + o) + o' = \mathsf{v}_i + (o + o')$. Sizes are partially ordered; size comparison $\boxed{a \leq b}$ holds as expected if either $b = \infty$ or $o \leq o'$ where either $a = o$ and $b = o'$ or $a \in \{o, \mathsf{v}_i + o\}$ and $b = \mathsf{v}_i + o'$. Different size variables are incomparable.

Fig. 2 lists the inductive and derived judgements of our type theory and figures 3 and 4 the inference rules. We have $\boxed{\text{boxed}}$ the rules dealing with irrelevant size application. Fig. 5 adds the typing and equality rules for case distinction and recursion on natural numbers. Judgement $\Gamma \vdash T : \mathsf{Adm}\, \ell$ characterizes the valid type annotations T in recursion $\mathsf{fix}_\ell\, T\, t$. The type constructor T has to be monotone in the size argument; this is a technical condition for type-based termination [Barthe et al. 2004]. We will make use of it in Section 4.7. We write $\boxed{\mathscr{D} :: J}$ to express that $\mathscr{D}$ is a derivation of judgement J.

In the typing judgement $\Gamma \vdash t : T$, the term t is in scope of Γ, i. e., may not mention irrelevant variables in relevant positions. However, the type T is in scope of the resurrected context $\Gamma^{\oplus}$, hence, can mention all variables declared in Γ. The other judgements are organized similarly. To understand this distinction, consider judgement $z \div \mathsf{Size} \vdash \mathsf{Nat}\, z$. This would mean that z is irrelevant in $\mathsf{Nat}\, z$

$\boxed{\vdash \Gamma}$ (implies $\vdash \Gamma^{\oplus}$) and $\boxed{\Gamma(i) = {\star}T}$ (implies $\Gamma^{\oplus} \vdash T$ if $\vdash \Gamma$).

$$\frac{}{\vdash ()} \qquad \frac{\vdash \Gamma \qquad \Gamma^{\oplus} \vdash T}{\vdash \Gamma.T} \qquad \frac{\vdash \Gamma}{\vdash \Gamma.{\star}\mathsf{Size}} \qquad \frac{}{(\Gamma.{\star}T)(0) = {\star}T\uparrow} \qquad \frac{\Gamma(i) = {\star}T}{(\Gamma.\_)(i+1) = {\star}T\uparrow}$$

$\boxed{\Gamma \vdash a : \mathsf{Size}}$ (implies $\vdash \Gamma$ and $\Gamma^{\oplus} \vdash a : \mathsf{Size}$).

$$\frac{\vdash \Gamma}{\Gamma \vdash \infty : \mathsf{Size}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash o : \mathsf{Size}}\, o \in \mathbb{N} \qquad \frac{\vdash \Gamma \qquad \Gamma(i) = {\div}\mathsf{Size}}{\Gamma \vdash \mathsf{v}_i + o : \mathsf{Size}}\, o \in \mathbb{N}$$

$\boxed{\Gamma \vdash t : T}$ (implies $\vdash \Gamma$ and $\Gamma^{\oplus} \vdash T$ [and $\Gamma^{\oplus} \vdash t : T$]. Note: no rule for $\Gamma \vdash \mathsf{Size} : s$.)

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Set}_{\ell} : \mathsf{Set}_{\ell'}}\,\ell{<}\ell' \qquad \frac{\Gamma \vdash U : s \qquad \Gamma.U \vdash T : s}{\Gamma \vdash \Pi\, U\, T : s} \qquad \frac{\Gamma.\mathsf{Size} \vdash T : s}{\Gamma \vdash \Pi^{\star}\mathsf{Size}\, T : s} \qquad \frac{\Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash \mathsf{Nat}\, a : \mathsf{Set}_0}$$

$$\frac{\vdash \Gamma \qquad \Gamma(i) = {\div}T}{\Gamma \vdash \mathsf{v}_i : T}\, T \neq \mathsf{Size} \qquad \frac{\Gamma.{\star}U \vdash t : T}{\Gamma \vdash \lambda t : \Pi^{\star}U\, T} \qquad \frac{\Gamma \vdash t : \Pi\, U\, T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\, u : T[u]}$$

$$\frac{\Gamma \vdash t : \Pi\, \mathsf{Size}\, T \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash t\, a : T[a]} \qquad \boxed{\frac{\Gamma \vdash t : \forall\, T \qquad \Gamma^{\oplus} \vdash a, b : \mathsf{Size}}{\Gamma \vdash t\langle a \rangle : T[b]}}$$

$$\frac{\Gamma^{\oplus} \vdash a, b : \mathsf{Size}}{\Gamma \vdash \mathsf{zero}\langle a \rangle : \mathsf{Nat}\,(b+1)} \qquad \frac{\Gamma^{\oplus} \vdash a : \mathsf{Size} \qquad \Gamma \vdash t : \mathsf{Nat}\, b}{\Gamma \vdash \mathsf{suc}\langle a \rangle t : \mathsf{Nat}\,(b+1)} \qquad \frac{\Gamma \vdash t : T \qquad \Gamma^{\oplus} \vdash T \leq T'}{\Gamma \vdash t : T'}$$

$\boxed{\Gamma \vdash T \leq T'}$ (implies $\Gamma \vdash T, T'$)

$$\frac{\vdash \Gamma \qquad \ell{\leq}\ell'}{\Gamma \vdash \mathsf{Set}_{\ell} \leq \mathsf{Set}_{\ell'}} \qquad \frac{\Gamma \vdash a \leq b : \mathsf{Size}}{\Gamma \vdash \mathsf{Nat}\, a \leq \mathsf{Nat}\, b} \qquad \frac{\Gamma \vdash T = T'}{\Gamma \vdash T \leq T'}$$

$$\frac{\Gamma \vdash U' \leq U \qquad \Gamma.U' \vdash T \leq T'}{\Gamma \vdash \Pi\, U\, T \leq \Pi\, U'\, T'} \qquad \frac{\Gamma.\mathsf{Size} \vdash T \leq T'}{\Gamma \vdash \Pi^{\star}\mathsf{Size}\, T \leq \Pi^{\star}\mathsf{Size}\, T'} \qquad \frac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3}$$

$\boxed{\Gamma \vdash \tau : \Delta}$ (implies $\vdash \Gamma$ and $\vdash \Delta$ [and $\Gamma^{\oplus} \vdash \tau : \Delta$] and $\Gamma^{\oplus} \vdash \tau : \Delta^{\oplus}$).

$$\frac{\vdash \Gamma}{\Gamma \vdash () : ()} \qquad \frac{\Gamma \vdash \tau : \Delta \qquad \Delta^{\oplus} \vdash T \qquad \Gamma \vdash t : T\tau}{\Gamma \vdash (\tau, t) : \Delta.T}$$

$$\frac{\Gamma \vdash \tau : \Delta \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash (\tau, a) : \Delta.\mathsf{Size}} \qquad \boxed{\frac{\Gamma \vdash \tau : \Delta \qquad \Gamma^{\oplus} \vdash a : \mathsf{Size}}{\Gamma \vdash (\tau, a) : \Delta.{\div}\mathsf{Size}}}$$

$\boxed{\Gamma \vdash \sigma = \sigma' \eqdot \tau : \Delta}$ (implies $\vdash \Gamma$ and $\vdash \Delta$ and $\Gamma \vdash \tau : \Delta$ and $\Gamma \vdash \sigma' = \sigma \eqdot \tau : \Delta$).

$$\frac{\vdash \Gamma}{\Gamma \vdash () = () \eqdot () : ()} \qquad \frac{\Gamma \vdash \sigma = \sigma' \eqdot \tau : \Delta \qquad \Delta^{\oplus} \vdash T \qquad \Gamma \vdash u = u' = t : T\tau}{\Gamma \vdash (\sigma, u) = (\sigma', u') \eqdot (\tau, t) : \Delta.T}$$

$$\frac{\Gamma \vdash \sigma = \sigma' \eqdot \tau : \Delta \qquad \Gamma \vdash a = a' = b : T\tau}{\Gamma \vdash (\sigma, a) = (\sigma', a') \eqdot (\tau, b) : \Delta.\mathsf{Size}} \qquad \boxed{\frac{\Gamma \vdash \sigma = \sigma' \eqdot \tau : \Delta \qquad \Gamma^{\oplus} \vdash a, a', b : T\tau}{\Gamma \vdash (\sigma, a) = (\sigma', a') \eqdot (\tau, b) : \Delta.{\div}\mathsf{Size}}}$$

Fig. 3. Typing, subtyping, and substitution judgements.

Computation rules.

$$\frac{\Gamma.U \vdash t : T \qquad \Gamma \vdash u : U}{\Gamma \vdash (\lambda t)\, u = t[u] : T[u]} \qquad \frac{\Gamma.\mathsf{Size} \vdash t : T \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash (\lambda t)\, a = t[a] : T[a]} \qquad \boxed{\frac{\Gamma.\dot{+}\mathsf{Size} \vdash t : T \qquad \Gamma^{\oplus} \vdash a, b : \mathsf{Size}}{\Gamma \vdash (\lambda t)\, \langle a \rangle = t[a] : T[b]}}$$

Extensionality rules.

$$\frac{\Gamma \vdash t : \Pi\, U\, T}{\Gamma \vdash t = \lambda x.t\, x : \Pi\, U\, T} \qquad \boxed{\frac{\Gamma \vdash t : \forall\, T \qquad \Gamma^{\oplus}.\mathsf{Size} \vdash a : \mathsf{Size}}{\Gamma \vdash t = \lambda x.t\langle a \rangle : \forall\, T}}$$

Congruence rules.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Set}_\ell = \mathsf{Set}_\ell : \mathsf{Set}_{\ell'}}\,\ell<\ell' \qquad \frac{\Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash \mathsf{Nat}\, a = \mathsf{Nat}\, a : \mathsf{Set}_0}$$

$$\frac{\Gamma \vdash U = U' : s \qquad \Gamma.U \vdash T = T' : s}{\Gamma \vdash \Pi\, U\, T = \Pi\, U'\, T' : s} \qquad \frac{\Gamma.\mathsf{Size} \vdash T = T' : s}{\Gamma \vdash \Pi^{\star}\mathsf{Size}\, T = \Pi^{\star}\mathsf{Size}\, T' : s}$$

$$\frac{\vdash \Gamma \qquad \Gamma(i) = \dot{} T}{\Gamma \vdash \mathsf{v}_i = \mathsf{v}_i : T}\, T \neq \mathsf{Size} \qquad \frac{\Gamma.^{\star}U \vdash t = t' : T}{\Gamma \vdash \lambda t = \lambda t' : \Pi^{\star} U\, T} \qquad \frac{\Gamma \vdash t = t' : \Pi\, U\, T \qquad \Gamma \vdash u = u' : U}{\Gamma \vdash t\, u = t'\, u' : T[u]}$$

$$\frac{\Gamma \vdash t = t' : \Pi\, \mathsf{Size}\, T \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash t\, a = t'\, a : T[a]} \qquad \boxed{\frac{\Gamma \vdash t = t' : \forall\, T \qquad \Gamma^{\oplus} \vdash a, a', b : \mathsf{Size}}{\Gamma \vdash t\langle a \rangle = t'\langle a' \rangle : T[b]}}$$

$$\frac{\Gamma^{\oplus} \vdash a, a', b : \mathsf{Size}}{\Gamma \vdash \mathsf{zero}\langle a \rangle = \mathsf{zero}\langle a' \rangle : \mathsf{Nat}\,(b+1)} \qquad \frac{\Gamma^{\oplus} \vdash a, a' : \mathsf{Size} \qquad \Gamma \vdash t = t' : \mathsf{Nat}\, b}{\Gamma \vdash \mathsf{suc}\langle a \rangle t = \mathsf{suc}\langle a' \rangle t' : \mathsf{Nat}\,(b+1)}$$

$$\frac{\Gamma \vdash t = t' : T \qquad \Gamma^{\oplus} \vdash T \leq T'}{\Gamma \vdash t = t' : T'}$$

Equivalence rules.

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t = t : T} \qquad \frac{\Gamma \vdash t = t' : T}{\Gamma \vdash t' = t : T} \qquad \frac{\Gamma \vdash t_1 = t_2 : T \qquad \Gamma \vdash t_2 = t_3 : T}{\Gamma \vdash t_1 = t_3 : T}$$

Fig. 4. Definitional equality $\boxed{\Gamma \vdash t = t' : T}$ (implies $\Gamma^{\oplus} \vdash T$ and $\Gamma \vdash t, t' : T$ [and $\Gamma^{\oplus} \vdash t = t' : T$]).

and thus, $\Gamma \vdash \mathsf{Nat}\, a = \mathsf{Nat}\, a'$ for all sizes $\Gamma^{\oplus} \vdash a, a' : \mathsf{Size}$. But this is exactly wrong! However, judgement $z \dot{+} \mathsf{Size} \vdash \mathsf{zero}\langle z \rangle : \mathsf{Nat}\,(z+1)$ is fine, it implies $\Gamma \vdash \mathsf{zero}\langle a \rangle = \mathsf{zero}\langle a' \rangle : \mathsf{Nat}\,(b+1)$ for all $\Gamma^{\oplus} \vdash a, a', b : \mathsf{Size}$.

Our substitution theorem needs to reflect the distinct scope of things left of the colon vs. things right of the colon. In the last example we have applied the substitution triple $\Gamma \vdash [a] = [a'] \coloneqq [b] : (z \dot{+} \mathsf{Size})$ to judgement $z \dot{+} \mathsf{Size} \vdash \mathsf{zero}\langle z \rangle : \mathsf{Nat}\,(z+1)$. The first two substitutions apply to the term side while the third substitution applies to the type side. The fact that we replace an irrelevant variable z allows a, a', b to refer to irrelevant variables from Γ, thus, they are in scope of $\Gamma^{\oplus}$.

Typing requires from annotations $\langle a \rangle$ in a term only that they are well-scoped size expressions, i. e., just mention relevant size variables. Let $\boxed{t^{\infty}}$ denote the *erasure* of term t, meaning that we replace all annotations $\langle a \rangle$ in t by $\langle \infty \rangle$. Let $\boxed{t \approx u}$ relate terms that only differ in their annotations, i. e., $t \approx u :\Longleftrightarrow t^{\infty} = u^{\infty}$. Erasure does not change the term modulo judgmental equality:

Case distinction.

$$\frac{\Gamma^{\oplus} \vdash T : \mathsf{Nat}\,(a{+}1) \to \mathsf{Set}_\ell \qquad \Gamma \vdash u : \mathsf{Nat}\,(a{+}1) \qquad \Gamma \vdash t_z : T\,(\mathsf{zero}\langle a\rangle) \qquad \Gamma \vdash t_s : (x : \mathsf{Nat}\,a) \to T\,(\mathsf{suc}\langle a\rangle x)}{\Gamma \vdash u\,\mathsf{case}_\ell\,T\,t_z\,t_s : T\,u}$$

$$\frac{\Gamma^{\oplus} \vdash T = T' : \mathsf{Nat}\,(a+1) \to \mathsf{Set}_\ell \qquad \Gamma \vdash u = u' : \mathsf{Nat}\,(a+1) \qquad \Gamma \vdash t_z = t_z' : T\,(\mathsf{zero}\langle a\rangle) \qquad \Gamma \vdash t_s = t_s' : (x : \mathsf{Nat}\,a) \to T\,(\mathsf{suc}\langle a\rangle x)}{\Gamma \vdash u\,\mathsf{case}_\ell\,T\,t_z\,t_s = u'\,\mathsf{case}_\ell\,T'\,t_z'\,t_s' : T\,u}$$

$$\frac{\Gamma^{\oplus} \vdash a, b : \mathsf{Size} \qquad \Gamma^{\oplus} \vdash T : \mathsf{Nat}\,(b+1) \to \mathsf{Set}_\ell \qquad \Gamma \vdash t_z : T\,(\mathsf{zero}\langle b\rangle) \qquad \Gamma \vdash t_s : (x : \mathsf{Nat}\,b) \to T\,(\mathsf{suc}\langle b\rangle x)}{\Gamma \vdash (\mathsf{zero}\langle a\rangle)\,\mathsf{case}_\ell\,T\,t_z\,t_s = t_z : T\,\mathsf{zero}\langle b\rangle}$$

$$\frac{\Gamma^{\oplus} \vdash a : \mathsf{Size} \quad \Gamma \vdash t : \mathsf{Nat}\,b \quad \Gamma^{\oplus} \vdash T : \mathsf{Nat}\,(b+1) \to \mathsf{Set}_\ell \qquad \Gamma \vdash t_z : T\,(\mathsf{zero}\langle b\rangle) \quad \Gamma \vdash t_s : (x : \mathsf{Nat}\,b) \to T\,(\mathsf{suc}\langle b\rangle x)}{\Gamma \vdash (\mathsf{suc}\langle a\rangle t)\,\mathsf{case}_\ell\,T\,t_z\,t_s = t_s\,t : T\,(\mathsf{suc}\langle b\rangle t)}$$

Recursion.

$$\frac{\Gamma \vdash u : \mathsf{Nat}\,a \qquad \Gamma^{\oplus} \vdash T : \mathsf{Adm}\ \ell \qquad \Gamma \vdash t : \mathsf{FixT}\,T}{\Gamma \vdash u\,\mathsf{fix}_\ell\,T\,t : T\,a\,u}$$

$$\frac{\Gamma \vdash u = u' : \mathsf{Nat}\,a \qquad \Gamma^{\oplus} \vdash T = T' : \mathsf{Adm}\ \ell \qquad \Gamma \vdash t = t' : \mathsf{FixT}\,T}{\Gamma \vdash u\,\mathsf{fix}_\ell\,T\,t = u'\,\mathsf{fix}_\ell\,T'\,t' : T\,a\,u}$$

$$\frac{\Gamma \vdash c : \mathsf{Nat}\,b \qquad \Gamma^{\oplus} \vdash a : \mathsf{Size} \qquad \Gamma^{\oplus} \vdash T : \mathsf{Adm}\ \ell \qquad \Gamma \vdash t : \mathsf{FixT}\,T}{\Gamma \vdash c\,\mathsf{fix}_\ell\,T\,t = t\langle a\rangle(\lambda x.x\,\mathsf{fix}_\ell\,T\,t)\,c : T\,b\,c}$$

Fig. 5. Rules for case distinction and recursion.

Lemma 3.1 (Erasure and similarity).

(1) *If* $\Gamma \vdash t : T$ *then* $\Gamma \vdash t = t^{\infty} : T$.
(2) *If* $\Gamma \vdash t, u : T$ *and* $t \approx u$ *then* $\Gamma \vdash t = u : T$.

We should remark here that we have *neither type unicity nor principal types* due to the irrelevant size application rule. In the following, we list syntactic properties of our judgements. To this end, let J match a part of a judgement.

Lemma 3.2 (Context well-formedness).

(1) *If* $\vdash \Gamma.\Delta$ *then* $\vdash \Gamma$
(2) *If* $\Gamma \vdash J$ *then* $\vdash \Gamma$.

All types in a context are considered in the resurrected context, which justifies the first statement of the following lemma. A resurrected context is more permissive, as it brings more variable into scope. As such, it is comparable to an extended context or a context where types have been replaced by subtypes. This intuition accounts for the remaining statements but (4). The latter is a defining property of substitutions: only replacement for irrelevant sizes may refer to irrelevant size variables.

Lemma 3.3 (Resurrection).

(1) $\vdash \Gamma$ *iff* $\vdash \Gamma^{\oplus}$. *Then* $\Gamma^{\oplus} \vdash \mathsf{id} : \Gamma$, *which can be written* $\mathsf{id} : \Gamma^{\oplus} \leq \Gamma$.
(2) *If* $\Gamma \vdash J$ *then* $\Gamma^{\oplus} \vdash J$.
(3) *If* $\Gamma \vdash \sigma : \Delta^{\oplus}$ *then* $\Gamma \vdash \sigma : \Delta$.
(4) *If* $\Gamma \vdash \sigma : \Delta$ *then* $\Gamma^{\oplus} \vdash \sigma : \Delta^{\oplus}$.

Lemma 3.4 (Substitution).

(1) *If* $\Gamma \vdash \sigma : \Delta$ *and* $\Delta \vdash J$ *then* $\Gamma \vdash J\sigma$.
(2) *If* $\Gamma \vdash \sigma = \sigma' \coloneqq \tau : \Delta$ *and* $\Delta \vdash t : T$ *then* $\Gamma \vdash t\sigma : T\tau$ *and* $\Gamma \vdash t\sigma' : T\tau$.

Lemma 3.5 (Specific substitutions).

(1) *If* $\vdash \Gamma.\Delta$ *then* $\Gamma.\Delta \vdash \uparrow^{|\Delta|}_{|\Gamma|} : \Gamma$. *If* $\vdash \Gamma.T$ *then* $\Gamma.T \vdash \uparrow : \Gamma$.
(2) *If* $\vdash \Gamma$ *then* $\Gamma \vdash \mathsf{id} : \Gamma$.
(3) *If* $\Gamma \vdash u : U$ *then* $\Gamma \vdash [u] : \Gamma.U$.

The relation $\Gamma \vdash \sigma = \sigma' \coloneqq \tau : \Delta$ is a partial equivalence relation (PER) on term-side substitutions σ, σ'. Note that usually we cannot resurrect this judgement to $\Gamma^{\oplus} \vdash \sigma = \sigma' \coloneqq \tau : \Delta^{\oplus}$. For instance, $z_1 \div \mathsf{Size}.\, z_2 \div \mathsf{Size} \vdash [z_1] = [z_2] \coloneqq [\infty] : z \div \mathsf{Size}$ holds but $z_1{:}\mathsf{Size}.\, z_2{:}\mathsf{Size} \vdash [z_1] = [z_2] \coloneqq [\infty] : z{:}\mathsf{Size}$ clearly not.

Lemma 3.6 (Substitution equality).

(1) *Conversion: If* $\Gamma \vdash \sigma = \sigma' \coloneqq \tau_1 : \Delta$ *and* $\Gamma^{\oplus} \vdash \tau_1 = \tau_2 \coloneqq \tau : \Delta^{\oplus}$ *then* $\Gamma \vdash \sigma = \sigma' \coloneqq \tau_2 : \Delta$.
(2) *Reflexivity: If* $\Gamma \vdash \sigma : \Delta$ *then* $\Gamma \vdash \sigma = \sigma \coloneqq \sigma : \Delta$.
(3) *Symmetry: If* $\Gamma \vdash \sigma = \sigma' \coloneqq \tau : \Delta$ *then* $\Gamma \vdash \sigma' = \sigma \coloneqq \tau : \Delta$.
(4) *Transitivity: If* $\Gamma \vdash \sigma_1 = \sigma_2 \coloneqq \tau : \Delta$ *and* $\Gamma \vdash \sigma_2 = \sigma_3 \coloneqq \tau : \Delta$ *then* $\Gamma \vdash \sigma_1 = \sigma_3 \coloneqq \tau : \Delta$.
(5) *Functionality: Let* $\Gamma \vdash \sigma = \sigma' \coloneqq \tau : \Delta$.
   (a) *If* $\Delta \vdash t : T$ *then* $\Gamma \vdash t\sigma = t\sigma' : T\tau$.
   (b) *If* $\Delta \vdash t = t' : T$ *then* $\Gamma \vdash t\sigma = t'\sigma' : T\tau$.
   (c) *Corollary: If* $\Delta \vdash T \leq T'$ *then* $\Gamma \vdash T\sigma \leq T\sigma'$.

Lemma 3.7 (Inversion of typing).

(1) *If* $\Gamma \vdash \mathsf{v}_i : T'$ *then* $\Gamma(i) = T$ *and* $\Gamma^{\oplus} \vdash T \leq T'$ *for some* T.
(2) *If* $\Gamma \vdash \lambda t : T'$ *then either* $\Gamma.U \vdash t : T$ *and* $\Gamma^{\oplus} \vdash \Pi\, U\, T \leq T'$ *for some* U, T *or* $\Gamma.^{\star}\mathsf{Size} \vdash t : T$ *and* $\Gamma^{\oplus} \vdash \Pi^{\star}\mathsf{Size}\, T \leq T'$ *for some* T.
(3) *If* $\Gamma \vdash t\, u : T'$ *then* $\Gamma \vdash t : \Pi\, U\, T$ *and* $\Gamma \vdash u : U$ *and* $\Gamma^{\oplus} \vdash T[u] \leq T'$ *for some* U, T.
(4) *If* $\Gamma \vdash t\, a : T'$ *then* $\Gamma \vdash t : \Pi\, \mathsf{Size}\, T$ *and* $\Gamma \vdash a : \mathsf{Size}$ *and* $\Gamma^{\oplus} \vdash T[a] \leq T'$ *for some* T.
(5) *If* $\Gamma \vdash t\, \langle a \rangle : T'$ *then* $\Gamma \vdash t : \forall\, T$ *and* $\Gamma^{\oplus} \vdash a, b : \mathsf{Size}$ *and* $\Gamma^{\oplus} \vdash T[b] \leq T'$ *for some* T, b.
(6) *... Analogous properties for the remaining term and type constructors.*

Proof. Each by induction on the typing derivation, gathering applications of the conversion rule via transitivity of subtyping. □

4 SEMANTICS AND COMPLETENESS OF NORMALIZATION BY EVALUATION

In this section we present an operational semantics of our language, define the NbE algorithm, construct a PER model, and demonstrate that NbE is complete for definitional equality, i. e., if $\Gamma \vdash t = t' : T$, then t and t' have the same normal form up to annotations.

$\mathsf{Ne} \ni m$	$::= \mathsf{v}_i \mid m\, v \mid m\, a \mid m\, \langle a \rangle \mid m\, \mathsf{case}_\ell\, V\, v_z\, v_s \mid m\, \mathsf{fix}_\ell\, V\, v$	neutral n.f.
$\mathsf{Nf} \ni v$	$::= m \mid \lambda v \mid \mathsf{zero}\langle a \rangle \mid \mathsf{suc}\langle a \rangle v \mid \mathsf{Set}_\ell \mid \mathsf{Nat}\, a \mid \Pi\, V_u\, V_t \mid \Pi^{\star}\mathsf{Size}\, V$	normal form

For the operational semantics, instead of defining a separate language of values, we extend the syntax of expressions by de Bruijn levels x_k to be used as generic values (unknowns), and type

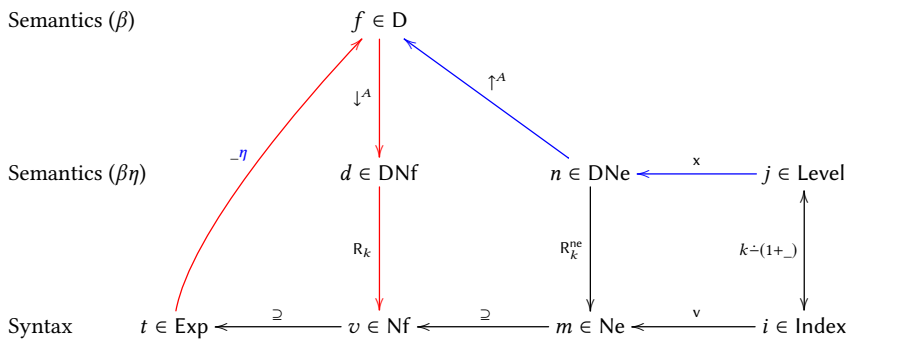

Fig. 6. Type-assignment NbE in locally nameless style.

annotations $\uparrow^A n$ and $\downarrow^A t$ for lazy realizations of the reflection and reification operations of NbE. *Terms* are expressions that do not contain these new expression forms. *Values* $\boxed{f, g, A, B, F \in \mathsf{D}}$ are expressions with no free de Bruijn indices, where each neutral n is under a reflection marker $\uparrow^A n$. The types A that direct reflection $\uparrow^A n$ and reification $\downarrow^A f$ also live in the value world.

NeExp	$\ni$	n	::=	$\cdots \mid \mathsf{x}_k$	de Bruijn level k
Up	$\ni$	N	::=	$\uparrow^A n$	reflection of neutral term n as value of type A
Whnf	$\ni$	w	::=	$\cdots \mid N$	reflected neutral is weak head normal
Exp	$\ni$	t	::=	$\cdots \mid \downarrow^A f$	reification of value f at type A

De Bruijn levels are the mirror images of de Bruijn indices. While de Bruijn indices index the context from the right, i. e., v_0 refers the last type that entered the context, de Bruijn levels index it from the left, i. e., x_0 refers to the first type in the context. This way, de Bruijn levels are stable under context extensions, and suitable to represent unknowns.

Size values $\boxed{\alpha, \beta \in \mathcal{S}ize}$ are size expressions that use de Bruijn levels instead of de Bruijn indices. Comparison of size values $\alpha \le \beta$ is analogous to comparison of size terms $a \le b$. In the following, we will reuse letter a for a value if it cannot be confused for a size term.

Finally, we identify two expression classes for NbE. Neutrals $n \in \mathsf{DNe}$ are the ones that will appear in values under the reflection marker $\uparrow^A$. Reified values $d \in \mathsf{DNf}$ are values under a reification marker $\downarrow^A$.

DNe	$\ni$	n	::=	$\mathsf{v}_i \mid n\,d \mid n\,\alpha \mid n\,\langle\alpha\rangle \mid n\,\mathsf{case}_\ell\,D\,d_z\,d_s \mid n\,\mathsf{fix}_\ell\,D\,d$	unreflected neutral value
DNf	$\ni$	d	::=	$\downarrow^A f$	reified value

Figure 6, adapted from Abel [2013] summarized the syntactic categories and main operations involved in NbE in what is called *locally nameless style*. The red path $\mathsf{Exp} \to \mathsf{D} \to \mathsf{D}_{\mathsf{nf}} \to \mathsf{Nf}$ decomposes $\beta\eta$-normalization into three steps.

(1) First, we close the term t with an environment η that maps the free de Bruijn indices of t to reflected de Bruijn levels. Reflection of de Bruijn levels follows the blue path $\mathsf{Level} \to \mathsf{DNe} \to \mathsf{D}$: Levels embed via constructor x into semantic neutrals DNe which are labeled with their type $A \in \mathsf{D}$ to become an element $\uparrow^A \mathsf{x}_j \in \mathsf{D}$.
(2) Then, we label value $t\eta \in \mathsf{D}$ with its type A to obtain $\downarrow^A t\eta \in \mathsf{DNf}$.

(3) Finally, *read back* $\mathsf{R}_k{\downarrow^A}t\eta$ produces a long normal form $v \in \mathsf{Nf}$, converting de Bruijn levels back to indices. Herein, k should be the length of the context the original term t lived in. If this is the case, each de Bruijn level encountered during read back is below k and can be safely converted to a de Bruijn index.

4.1 Weak Head Reduction

We define the operational semantics of our language by the weak head evaluation relation $\boxed{t \searrow w}$ which is defined on expressions, thus works on values as well as on terms. It is defined mutually with auxiliary relation $\boxed{w \mathbin{@} e \searrow w'}$ stating that weak head normal form w is eliminated by e into weak head normal form w'.

$\boxed{t \searrow w}$ and $\boxed{w \mathbin{@} e \searrow w'}$

$$\frac{}{w \searrow w} \qquad \frac{t \searrow w \qquad w \mathbin{@} e \searrow w'}{t\,e \searrow w'}$$

$$\frac{t[u] \searrow w}{(\lambda t) \mathbin{@} u \searrow w} \qquad \frac{t[\alpha] \searrow w}{(\lambda t) \mathbin{@} \alpha \searrow w} \qquad \frac{t[\alpha] \searrow w}{(\lambda t) \mathbin{@} \langle\alpha\rangle \searrow w} \qquad \frac{t_z \searrow w}{(\mathsf{zero}\langle\alpha\rangle) \mathbin{@} \mathsf{case}_\ell\, T\, t_z\, t_s \searrow w}$$

$$\frac{t_s\, t \searrow w}{(\mathsf{suc}\langle\alpha\rangle t) \mathbin{@} \mathsf{case}_\ell\, T\, t_z\, t_s \searrow w} \qquad \frac{t\,\langle\alpha\rangle\,(\lambda x.\, x\,\mathsf{fix}_\ell\, T\, t)\, c \searrow w}{c \mathbin{@} \mathsf{fix}_\ell\, T\, t \searrow w}\; c \in \{\mathsf{zero}\langle\alpha\rangle, \mathsf{suc}\langle\alpha\rangle u\}$$

For NbE, we add evaluation rules that deal with elimination of delayed reflection:

$$\frac{A' \searrow \Pi\, A\, B}{(\uparrow^{A'} n) \mathbin{@} u \searrow \uparrow^{B[u]} (n {\downarrow^A} u)} \qquad \frac{A \searrow \Pi\, \mathsf{Size}\, B}{(\uparrow^A n) \mathbin{@} \alpha \searrow \uparrow^{B[\alpha]} (n\,\alpha)} \qquad \frac{A \searrow \forall\, B}{(\uparrow^A n) \mathbin{@} \langle\alpha\rangle \searrow \uparrow^{B[\alpha]} (n\langle\alpha\rangle)}$$

$$\frac{}{(\uparrow^A n) \mathbin{@} \mathsf{case}_\ell\, B\, f_z\, f_s \searrow \uparrow^{B\,(\uparrow^A n)} n\, \mathsf{case}_\ell\, (\downarrow^{\mathsf{Nat}\,\infty \to \mathsf{Set}_\ell} B)\, (\downarrow^{B\,\mathsf{zero}\langle\infty\rangle} f_z)\, (\downarrow^{(x:\mathsf{Nat}\,\infty) \to B\,(\mathsf{suc}\langle\infty\rangle x)} f_s)}$$

$$\frac{}{(\uparrow^A n) \mathbin{@} \mathsf{fix}_\ell\, B\, f \searrow n\, \mathsf{fix}_\ell\, (\downarrow^{\mathsf{FixK}} B)\, (\downarrow^{\mathsf{FixT}\, B} f)}$$

4.2 Read Back

The *read back* phase of NbE [Grégoire and Leroy 2002] transforms a reified value d into a normal form v. It is specified via an inductively defined relation $\mathsf{R}_k\, d \searrow v$ and several auxiliary relations. The number k, will be instantiated by the length of the context Γ later. It allows us to transform a de Bruijn level l into a de Bruijn index i, via the law $i + l + 1 = k$. At this point, we do not ensure that the k is large enough to accommodate the de Bruijn levels in d. Levels $l \geq k$ which are to big will simply be mapped to de Bruijn index 0. The correct k is later ensured by our logical relation (Section 5). Even though read back operates on values in practice, formally it is defined on expressions.

$\boxed{\mathsf{R}_k\, d \searrow v}$ Read back reified value d.

$$\frac{U \searrow s \qquad \mathsf{R}^{\mathsf{ty}}_k\, T \searrow V}{\mathsf{R}_k {\downarrow^U} T \searrow V} \qquad \frac{U \searrow \mathsf{Nat}\,\alpha \qquad \mathsf{R}^{\mathsf{nat}}_k\, u \searrow v}{\mathsf{R}_k {\downarrow^U} u \searrow v} \qquad \frac{U \searrow N \qquad \mathsf{R}^{\mathsf{up}}_k\, u \searrow m}{\mathsf{R}_k {\downarrow^U} u \searrow m}$$

$$\frac{U \searrow \Pi\, A\, B \qquad \mathsf{R}_{k+1} {\downarrow^{B[\uparrow^A \mathsf{x}_k]}} (f \uparrow^A \mathsf{x}_k) \searrow v}{\mathsf{R}_k {\downarrow^U} f \searrow \lambda v}$$

$$\frac{U \searrow \Pi\, \mathsf{Size}\, B \qquad \mathsf{R}_{k+1} {\downarrow^{B[\mathsf{x}_k]}} (f\, \mathsf{x}_k) \searrow v}{\mathsf{R}_k {\downarrow^U} f \searrow \lambda v} \qquad \frac{U \searrow \forall\, B \qquad \mathsf{R}_{k+1} {\downarrow^{B[\mathsf{x}_k]}} (f\, \langle \mathsf{x}_k \rangle) \searrow v}{\mathsf{R}_k {\downarrow^U} f \searrow \lambda v}$$

$\boxed{\mathsf{R}^{\mathsf{up}}_k\ t \searrow m}$ Read back neutrals under annotation. (The annotation is ignored.)

$$\frac{t \searrow \uparrow^T n \qquad \mathsf{R}^{\mathsf{ne}}_k\ n \searrow m}{\mathsf{R}^{\mathsf{up}}_k\ t \searrow m}$$

$\boxed{\mathsf{R}^{\mathsf{nat}}_k\ t \searrow v}$ Read back natural number value.

$$\frac{\mathsf{R}^{\mathsf{up}}_k\ t \searrow m}{\mathsf{R}^{\mathsf{nat}}_k\ t \searrow m} \qquad \frac{t \searrow \mathsf{zero}\langle\alpha\rangle \qquad \mathsf{R}^{\mathsf{size}}_k\ \alpha \searrow a}{\mathsf{R}^{\mathsf{nat}}_k\ t \searrow \mathsf{zero}\langle a\rangle} \qquad \frac{t \searrow \mathsf{suc}\langle\alpha\rangle u \qquad \mathsf{R}^{\mathsf{size}}_k\ \alpha \searrow a \qquad \mathsf{R}^{\mathsf{nat}}_k\ u \searrow v}{\mathsf{R}^{\mathsf{nat}}_k\ t \searrow \mathsf{suc}\langle a\rangle v}$$

$\boxed{\mathsf{R}^{\mathsf{size}}_k\ \alpha \searrow a}$ Read back size value α.

$$\frac{}{\mathsf{R}^{\mathsf{size}}_k\ \infty \searrow \infty} \qquad \frac{}{\mathsf{R}^{\mathsf{size}}_k\ o \searrow o} \qquad \frac{}{\mathsf{R}^{\mathsf{size}}_k\ \mathsf{x}_j + o \searrow \mathsf{v}_{k \dot{-} (1+j)} + o}$$

$\boxed{\mathsf{R}^{\mathsf{ne}}_k\ n \searrow m}$ and $\boxed{\mathsf{R}^{\mathsf{elim}}_k\ e \searrow e^v}$ Read back unreflected neutral.

$$\frac{\mathsf{R}^{\mathsf{elim}}_k\ e_i \searrow e_i^v \text{ for all } i}{\mathsf{R}^{\mathsf{ne}}_k\ \mathsf{x}_j\ \vec{e} \searrow \mathsf{v}_{k \dot{-} (1+j)}\ \vec{e}^v} \qquad \frac{\mathsf{R}_k\ d \searrow v}{\mathsf{R}^{\mathsf{elim}}_k\ d \searrow v} \qquad \frac{\mathsf{R}^{\mathsf{size}}_k\ \alpha \searrow b}{\mathsf{R}^{\mathsf{elim}}_k\ \alpha \searrow b} \qquad \frac{\mathsf{R}^{\mathsf{size}}_k\ \alpha \searrow b}{\mathsf{R}^{\mathsf{elim}}_k\ \langle\alpha\rangle \searrow \langle b\rangle}$$

$$\frac{\mathsf{R}_k\ D \searrow V \qquad \mathsf{R}_k\ d_z \searrow v_z \qquad \mathsf{R}_k\ d_s \searrow v_s}{\mathsf{R}^{\mathsf{elim}}_k\ (\mathsf{case}_\ell\ D\ d_z\ d_s) \searrow \mathsf{case}_\ell\ V\ v_z\ v_s} \qquad \frac{\mathsf{R}_k\ D \searrow V \qquad \mathsf{R}_k\ d \searrow v}{\mathsf{R}^{\mathsf{elim}}_k\ (\mathsf{fix}_\ell\ D\ d) \searrow \mathsf{fix}_\ell\ V\ v}$$

$\boxed{\mathsf{R}^{\mathsf{ty}}_k\ T \searrow V}$ Read back type value.

$$\frac{T \searrow \mathsf{Set}_\ell}{\mathsf{R}^{\mathsf{ty}}_k\ T \searrow \mathsf{Set}_\ell} \qquad \frac{T \searrow \mathsf{Nat}\ \alpha \qquad \mathsf{R}^{\mathsf{size}}_k\ \alpha \searrow b}{\mathsf{R}^{\mathsf{ty}}_k\ T \searrow \mathsf{Nat}\ b} \qquad \frac{\mathsf{R}^{\mathsf{up}}_k\ T \searrow m}{\mathsf{R}^{\mathsf{ty}}_k\ T \searrow m}$$

$$\frac{T \searrow \Pi\ A\ B \qquad \mathsf{R}^{\mathsf{ty}}_k\ A \searrow V_a \qquad \mathsf{R}^{\mathsf{ty}}_{k+1}\ B \searrow V_b}{\mathsf{R}^{\mathsf{ty}}_k\ T \searrow \Pi\ V_a\ V_b} \qquad \frac{T \searrow \Pi^\star \mathsf{Size}\ B \qquad \mathsf{R}^{\mathsf{ty}}_{k+1}\ B \searrow V}{\mathsf{R}^{\mathsf{ty}}_k\ T \searrow \Pi^\star \mathsf{Size}\ V}$$

4.3 Partial Equivalence Relations

A type T will be interpreted as a partial equivalence relation (PER) $\mathcal{A}$ on terms, i. e., a relation which is symmetric and transitive. The domain $\mathrm{dom}(\mathcal{A})$ of the relation can be thought of as the set of terms which denotes the extension of the type; on $\mathrm{dom}(\mathcal{A}) = \{a \mid \exists a'.\ (a, a') \in \mathcal{A}\}$ the relation $\mathcal{A}$ is in fact an equivalence relation. We write $a = a' \in \mathcal{A}$ for relatedness in $\mathcal{A}$ and $a \in \mathcal{A}$ if $a \in \mathrm{dom}(\mathcal{A})$.

The PERs $\mathcal{N}e$ and $\mathcal{N}f$ characterize (neutral) normalizing values. For instance, two values n and n' are related in $\mathcal{N}e$ if at any $k \in \mathbb{N}$ they can be read back to neutral normal forms m and m' which are identical up to annotations.

$$\begin{array}{lcl}
\boxed{n = n' \in \mathcal{N}e} & :\Longleftrightarrow & \mathsf{R}^{\mathsf{ne}}_k\ n \searrow m \text{ and } \mathsf{R}^{\mathsf{ne}}_k\ n' \searrow m' \text{ and } m \approx m' \text{ for all } k \\
\boxed{d = d' \in \mathcal{N}f} & :\Longleftrightarrow & \mathsf{R}_k\ d \searrow v \text{ and } \mathsf{R}_k\ d' \searrow v' \text{ and } v \approx v' \text{ for all } k \\
\boxed{e = e' \in \mathscr{E}lim} & :\Longleftrightarrow & \mathsf{R}^{\mathsf{elim}}_k\ e \searrow e_v \text{ and } \mathsf{R}^{\mathsf{elim}}_k\ e' \searrow e'_v \text{ and } e_v \approx e'_v \text{ for all } k \\
\boxed{A = A' \in \mathcal{T}y} & :\Longleftrightarrow & \mathsf{R}^{\mathsf{ty}}_k\ A \searrow V \text{ and } \mathsf{R}^{\mathsf{ty}}_k\ A' \searrow V' \text{ and } V \approx V' \text{ for all } k
\end{array}$$

Once we have established useful closure properties of these PERs, they abstract most of the reasoning about the read-back relation from our proofs. This idea is due to Coquand [Abel et al. 2009].

Lemma 4.1 (Closure properties of $\mathcal{N}e$).
(1) $\mathsf{x}_k = \mathsf{x}_k \in \mathcal{N}e$.
(2) *If* $n = n' \in \mathcal{N}e$ *and* $e = e' \in \mathscr{C}lim$ *then* $n\,e = n'\,e' \in \mathcal{N}e$.

Lemma 4.2 (Closure properties of $\mathscr{C}lim$).
(1) *If* $d = d' \in \mathcal{N}f$ *then* $d = d' \in \mathscr{C}lim$.
(2) *If* $\alpha \in \mathscr{S}ize$ *then* $\alpha = \alpha \in \mathscr{C}lim$.
(3) *If* $\alpha, \alpha' \in \mathscr{S}ize$ *then* $\langle\alpha\rangle = \langle\alpha'\rangle \in \mathscr{C}lim$.
(4) *If* $A = A' \in \mathcal{T}y$ *and* $d_z{=}d'_z \in \mathcal{N}f$ *and* $d_s{=}d'_s \in \mathcal{N}f$ *then* $\mathsf{case}_\ell\, A\, d_z\, d_s = \mathsf{case}_\ell\, A'\, d'_z\, d'_s \in \mathscr{C}lim$.
(5) *If* $D = D' \in \mathcal{N}f$ *and* $d = d' \in \mathcal{N}f$ *then* $\mathsf{fix}_\ell\, D\, d = \mathsf{fix}_\ell\, D'\, d' \in \mathscr{C}lim$.

Now we define some PERs and PER constructors on values. All these PERs $\mathscr{A}$ are closed under weak head equality, meaning if $a = b \in \mathscr{A}$ and a' has the same weak head normal form as a, then $a' = b \in \mathscr{A}$. (By symmetry, $\mathscr{A}$ is also closed under weak head equality on the second argument.)

PER $\mathcal{N}\mathscr{E}$ interprets all neutral types.

$$\boxed{t = t' \in \mathcal{N}\mathscr{E}} :\Longleftrightarrow t \searrow \uparrow^T n \text{ and } t' \searrow \uparrow^{T'} n' \text{ and } n = n' \in \mathcal{N}e.$$

$\boxed{\mathcal{N}at(\alpha)}$ interprets Nat α and is defined inductively by the following rules.

$$\frac{t = t' \in \mathcal{N}\mathscr{E}}{t = t' \in \mathcal{N}at(\beta)} \qquad \frac{t \searrow \mathsf{zero}\langle\alpha\rangle \quad t' \searrow \mathsf{zero}\langle\alpha'\rangle}{t = t' \in \mathcal{N}at(\beta+1)} \qquad \frac{t \searrow \mathsf{suc}\langle\alpha\rangle u \quad t' \searrow \mathsf{suc}\langle\alpha'\rangle u' \quad u = u' \in \mathcal{N}at(\beta)}{t = t' \in \mathcal{N}at(\beta+1)}$$

$\boxed{\mathscr{S}ize}$ interprets Size and is a discrete PER of size values:

$$\frac{}{\infty = \infty \in \mathscr{S}ize} \qquad \frac{}{o = o \in \mathscr{S}ize} \qquad \frac{}{\mathsf{x}_k + o = \mathsf{x}_k + o \in \mathscr{S}ize}$$

Let $\mathscr{A}$ be a PER (including $\mathscr{A} = \mathscr{S}ize$) and $\mathscr{F}$ a family of PERs over $\mathscr{A}$ such that $\mathscr{F}(u) = \mathscr{F}(u')$ whenever $u = u' \in \mathscr{A}$. We define

$$\boxed{\textstyle\prod \mathscr{A}\mathscr{F}} :\Longleftrightarrow \{(t,t') \mid t\,u = t'\,u' \in \mathscr{F}(u) \text{ for all } u = u' \in \mathscr{A}\}.$$

For a family $\mathscr{F}$ over $\mathscr{S}ize$ we also have the irrelevant function space

$$\boxed{\forall\,\mathscr{F}} \quad :\Longleftrightarrow \{(t,t') \mid t\langle\alpha\rangle = t'\langle\alpha'\rangle \in \mathscr{F}(\beta) \text{ for all } \alpha, \alpha', \beta \in \mathscr{S}ize\}.$$

4.4 PER Model

Semantic types and their interpretation as PERs are now defined via a family of inductive-recursive definitions [Dybjer 2000], one for each universe level ℓ. The construction follows Abel et al. [2007].

By induction on $\ell \in \mathbb{N}$ we define the PER family $\_ = \_ \in \mathsf{Set}_\ell$ of types together with the extension $\mathscr{E}\ell_\ell T$ (for $T = T' \in \mathsf{Set}_\ell$) which is a PER of values of type T. The rules for $\boxed{T = T' \in \mathscr{S}et_\ell}$ are listed in Fig. 7. All relations involved here are closed under weak head equality.

Lemma 4.3 (Well-definedness). *Let* $\mathscr{D} :: T_1 = T_2 \in \mathscr{S}et_\ell$.
(1) *Symmetry:* $T_2 = T_1 \in \mathscr{S}et_\ell$.
(2) *Transitivity: If* $T_2 = T_3 \in \mathscr{S}et_\ell$ *then* $T_1 = T_3 \in \mathscr{S}et_\ell$.
(3) *Extension:* $\mathscr{E}\ell_\ell(T_1) = \mathscr{E}\ell_\ell(T_2)$ *and "both" are PERs.*

Lemma 4.4 (Derivation independence of extension). *If* $\mathscr{D}_1 :: T = T_1 \in \mathscr{S}et_{\ell_1}$ *and* $\mathscr{D}_2 :: T_2 = T \in \mathscr{S}et_{\ell_2}$ *then* $\mathscr{E}\ell_{\ell_1}(T) = \mathscr{E}\ell_{\ell_2}(T)$.

Since $\mathscr{E}\ell_\ell(T)$ does not depend on ℓ nor the derivation that introduced $T = T' \in \mathscr{S}et_\ell$, we may simply write $t = t' \in \mathscr{E}\ell(T)$ or even $t = t' \in T$.

$$\frac{T = T' \in \mathcal{NE}}{T = T' \in \mathcal{S}et_\ell} \qquad \mathscr{E}\ell_\ell(T) = \mathcal{NE}$$

$$\frac{T \searrow \mathsf{Nat}\ \alpha \qquad T' \searrow \mathsf{Nat}\ \alpha}{T = T' \in \mathcal{S}et_\ell} \qquad \mathscr{E}\ell_\ell(T) = \mathcal{N}at(\alpha)$$

$$\frac{T \searrow \mathsf{Set}_{\ell'} \qquad T' \searrow \mathsf{Set}_{\ell'}}{T = T' \in \mathcal{S}et_\ell}\ \ell'<\ell \qquad \mathscr{E}\ell_\ell(T) = \mathcal{S}et_{\ell'}$$

$$\frac{\begin{array}{l} T \searrow \Pi\, A\, B \qquad T' \searrow \Pi\, A'\, B' \qquad A = A' \in \mathcal{S}et_\ell \\ B[u] = B'[u'] \in \mathcal{S}et_\ell \text{ for all } u = u' \in \mathscr{E}\ell_\ell(A) \end{array}}{T = T' \in \mathcal{S}et_\ell} \qquad \mathscr{E}\ell_\ell(T) = \prod(\mathscr{E}\ell_\ell(A),\ u \mapsto \mathscr{E}\ell_\ell(B[u]))$$

$$\frac{\begin{array}{l} T \searrow \Pi\, \mathsf{Size}\, B \qquad T' \searrow \Pi\, \mathsf{Size}\, B' \\ B[\alpha] = B'[\alpha] \in \mathcal{S}et_\ell \text{ for all } \alpha \in \mathcal{S}ize \end{array}}{T = T' \in \mathcal{S}et_\ell} \qquad \mathscr{E}\ell_\ell(T) = \prod(\mathcal{S}ize,\ \alpha \mapsto \mathscr{E}\ell_\ell(B[\alpha]))$$

$$\frac{\begin{array}{l} T \searrow \forall\, B \qquad T' \searrow \forall\, B' \\ B[\alpha] = B'[\alpha] \in \mathcal{S}et_\ell \text{ for all } \alpha \in \mathcal{S}ize \end{array}}{T = T' \in \mathcal{S}et_\ell} \qquad \mathscr{E}\ell_\ell(T) = \forall(\alpha \mapsto \mathscr{E}\ell_\ell(B[\alpha]))$$

Fig. 7. Semantic types and their interpretation.

4.5 Subtyping

The semantic types (PERs) admit subsumption:

Lemma 4.5 (Subsumption).

(1) *If* $\alpha \le \beta$ *then* $\mathcal{N}at(\alpha) \subseteq \mathcal{N}at(\beta)$.
(2) *If* $\mathcal{F}(\alpha) \subseteq \mathcal{F}'(\alpha)$ *for all* $\alpha \in \mathcal{S}ize$, *then* $\forall\mathcal{F} \subseteq \forall\mathcal{F}'$.
(3) *If* $\mathcal{A}' \subseteq \mathcal{A}$ *and* $\mathcal{F}(u) \subseteq \mathcal{F}'(u)$ *for all* $u \in \mathcal{A}'$, *then* $\prod \mathcal{A}\mathcal{F} \subseteq \prod \mathcal{A}'\mathcal{F}'$.
(4) *If* $\ell \le \ell'$ *then* $\mathcal{S}et_\ell \subseteq \mathcal{S}et_{\ell'}$.

We define subtyping of type values $\boxed{T \le T' \in \mathcal{T}ype}$ by induction on $T \in \mathcal{S}et_\ell$ and $T' \in \mathcal{S}et_{\ell'}$. Simultaneously, we need to prove correctness, namely that $T \le T' \in \mathcal{T}ype$ implies $\mathscr{E}\ell(T) \subseteq \mathscr{E}\ell(T')$. The correctness follows from Lemma 4.5 and we do not spell it out here.

$$\frac{T = T' \in \mathcal{NE}}{T \le T' \in \mathcal{T}ype} \qquad \frac{T \searrow \mathsf{Nat}\ \alpha \qquad T' \searrow \mathsf{Nat}\ \alpha' \qquad \alpha \le \alpha'}{T \le T' \in \mathcal{T}ype} \qquad \frac{T \searrow \mathsf{Set}_{\ell_0} \qquad T' \searrow \mathsf{Set}_{\ell_0'} \qquad \ell_0 \le \ell_0'}{T \le T' \in \mathcal{T}ype}$$

$$\frac{T \searrow \Pi\, A\, B \qquad T' \searrow \Pi\, A'\, B' \qquad A' \le A \in \mathcal{T}ype \qquad B[u] \le B'[u'] \in \mathcal{T}ype \text{ for all } u = u' \in A'}{T \le T' \in \mathcal{T}ype}$$

$$\frac{T \searrow \Pi^\star\, \mathsf{Size}\, B \qquad T' \searrow \Pi^\star\, \mathsf{Size}\, B' \qquad B[\alpha] \le B'[\alpha] \in \mathcal{T}ype \text{ for all } \alpha \in \mathcal{S}ize}{T \le T' \in \mathcal{T}ype}$$

Lemma 4.6 (Subtyping is a preorder).

(1) *If* $T = T' \in \mathcal{S}et_\ell$ *then* $T \le T' \in \mathcal{T}ype$.
(2) *If* $T_1 \le T_2 \in \mathcal{T}ype$ *and* $T_2 \le T_3 \in \mathcal{T}ype$ *then* $T_1 \le T_3 \in \mathcal{T}ype$.

$$\frac{T \searrow N \qquad S \searrow N'}{T \sqsubseteq S} \qquad \frac{T \searrow \Pi\, A\, B \qquad S \searrow \Pi\, A'\, B' \qquad A \sqsubseteq A' \qquad B[u] \sqsubseteq B'[u'] \text{ for all } u = u' \in A}{T \sqsubseteq S}$$

$$\frac{T \searrow \mathrm{Set}_\ell \qquad S \searrow \mathrm{Set}_\ell}{T \sqsubseteq S} \qquad \frac{T \searrow \Pi\,\mathrm{Size}\, B \qquad S \searrow \Pi\,\mathrm{Size}\, B' \qquad B[\alpha] \sqsubseteq B'[\alpha] \text{ for all } \alpha \in \mathit{Size}}{T \sqsubseteq S}$$

$$\frac{T \searrow \mathrm{Nat}\,\alpha \qquad S \searrow \mathrm{Nat}\,\beta}{T \sqsubseteq S} \qquad \frac{T \searrow \forall\, B \qquad S \searrow \forall\, B' \qquad B[\alpha] \sqsubseteq B'[\alpha'] \text{ for all } \alpha, \alpha' \in \mathit{Size}}{T \sqsubseteq S}$$

Fig. 8. Type shapes $\boxed{T \sqsubseteq S}$.

4.6 Type Shapes

Reflection and reification perform η-expansion so that we arrive at an η-long β-normal form. To perform the η-expansion, the precise type is not needed, just the approximate shape, in particular, whether it is a function type (do expand) or a base type (do not expand). For the logical framework, the shape of a dependent type is just its underlying simple type [Harper and Pfenning 2005]. However, in the presence of universes and large eliminations, there is no underlying simple type. Of course, we can take a type as its own shape, but we want at least that $\mathrm{Nat}\,\alpha$ and $\mathrm{Nat}\,\beta$ have the same shape even for different α, β. Also all neutral types can be summarized under a single shape.

We make our intuition precise by defining a relation $T \sqsubseteq S$ between type values, to express that S is a possible shape of type T. The asymmetry of this relation stems from the case for function types. At function types $\Pi\, A\, B \sqsubseteq \Pi\, R\, S$, we take S to be a family over domain A, not R! We cannot take R since we have to compare families B and S at a common domain, and A and R are not equal.

Fig. 8 defines $\boxed{T \sqsubseteq S}$ for $T \in \mathit{Set}_\ell$. We call T the *template* and S one of its possible *shapes*. Note that $T \in \mathit{Set}_\ell$ and $T \sqsubseteq S$ do not imply $S \in \mathit{Set}_\ell$. Type shapes are not well-defined types in general. For instance, assume a term $F : \mathrm{Nat}\,0 \to \mathrm{Set}_0$ which diverges if applied to a successor term. Then $T := (x : \mathrm{Nat}\,0) \to F\,x$ is a well-defined type; we have $T \in \mathit{Set}_0$. Now consider $S := (x : \mathrm{Nat}\,\infty) \to F\,x$. We have $T \sqsubseteq S$, but S is not well-defined; $S \notin \mathit{Set}_0$.

Lemma 4.7 (Types are their own shapes). *If $T = T' \in \mathit{Set}_\ell$ then $T \sqsubseteq T'$.*

Lemma 4.8 (Templates are up to equality). *If $T = T' \in \mathit{Set}_\ell$ and $T' \sqsubseteq S$ then $T \sqsubseteq S$.*

However, templates are not closed under subtyping in either direction because subtyping is contravariant for function type domains but the shape relation is covariant.

Further, it is not true that equal types make equally good shapes. We do not have that $T \sqsubseteq S$ and $S = S' \in \mathit{Set}_\ell$ imply $T \sqsubseteq S'$. This property fails for function types. Given $\Pi\, U\, T \sqsubseteq \Pi\, R\, S$ and $\Pi\, R\, S = \Pi\, R'\, S' \in \mathit{Set}_\ell$ we would need to show that $T[u] \sqsubseteq S'[u']$ for all $u = u' \in U$, but we only have $S[u] = S'[u'] \in \mathit{Set}_\ell$ for all $u = u' \in R$, thus the induction does not go through. The fact that $U \sqsubseteq R$ does not give us a handle on their inhabitants, we would need a stronger relation such as $U \leq R \in \mathcal{T}\mathit{ype}$. It is possible to construct an actual counterexample, using $\Pi\, R'\, S' = (x : \mathrm{Nat}\,0) \to F\,x$ from above and $\Pi\, R\, S = (x : \mathrm{Nat}\,0) \to G\,x$ such that G is defined on all of $\mathrm{Nat}\,\infty$ but agrees with F only on $x \in \mathrm{Nat}\,0$. Then $\Pi\, U\, T = (x : \mathrm{Nat}\,\infty) \to G\,x$ gives the desired counterexample.

Shapes are used to direct η-expansion when we reflect neutrals into semantic types and reify semantic values to long normal forms. The following theorem is the heart of our technical development.

Theorem 4.9 (Reflection and reification). *Let $T \in \mathit{Set}_\ell$ and $T \sqsubseteq S_1$ and $T \sqsubseteq S_2$.*

(1) *If* $n_1 = n_2 \in \mathcal{N}e$ *then* $\uparrow^{S_1} n_1 = \uparrow^{S_2} n_2 \in T$.
(2) *If* $t_1 = t_2 \in T$ *then* $\downarrow^{S_1} t_1 = \downarrow^{S_2} t_2 \in \mathcal{N}f$.

Proof. By induction on $T \in \mathcal{S}et_\ell$ and cases on $T \sqsubseteq S_1$ and $T \sqsubseteq S_2$.
Case $T \searrow \forall\, B$ with $B[\alpha] \in \mathcal{S}et_\ell$ for all $\alpha \in \mathcal{S}ize$

$$\frac{S_1 \searrow \forall\, B_1 \qquad B[\alpha] \sqsubseteq B_1[\alpha'] \text{ for all } \alpha, \alpha' \in \mathcal{S}ize}{T \sqsubseteq S_1}$$

$$\frac{S_2 \searrow \forall\, B_2 \qquad B[\alpha] \sqsubseteq B_2[\alpha'] \text{ for all } \alpha, \alpha' \in \mathcal{S}ize}{T \sqsubseteq S_2}$$

(1) To show $\uparrow^{S_1} n_1 = \uparrow^{S_2} n_2 \in T$ assume arbitrary $\alpha_1, \alpha_2, \beta \in \mathcal{S}ize$. Since $n_1\langle\alpha_1\rangle = n_2\langle\alpha_2\rangle \in \mathcal{N}e$ by Lemma 4.1, we obtain $\uparrow^{B_1[\alpha_1]}(n_1\langle\alpha_1\rangle) = \uparrow^{B_2[\alpha_2]}(n_2\langle\alpha_2\rangle) \in B[\beta]$ by induction hypothesis. Thus, $(\uparrow^{S_1} n_1)\langle\alpha_1\rangle = (\uparrow^{S_2} n_2)\langle\alpha_2\rangle \in B[\beta]$ by weak head expansion, which entails the goal by definition of $\mathscr{C}\ell(T)$.
(2) Assume $k \in \mathbb{N}$ and note that $\mathsf{x}_k = \mathsf{x}_k \in \mathcal{S}ize$, hence, $t_1\,\langle \mathsf{x}_k\rangle = t_2\,\langle \mathsf{x}_k\rangle \in B[\mathsf{x}_k]$. Thus, by induction hypothesis, $\mathsf{R}_{k+1}\downarrow^{B_i[\mathsf{x}_k]}(t_i\,\langle \mathsf{x}_k\rangle) \searrow v_i$ with $v_1 \approx v_2$, and finally $\mathsf{R}_k \downarrow^{S_i} t_i \searrow \lambda v_i$ by definition of read back. □

Corollary 4.10. *Let* $T \in \mathcal{S}et_\ell$.

(1) *If* $n = n' \in \mathcal{N}e$ *then* $\uparrow^T n = \uparrow^T n' \in T$.
(2) *If* $t = t' \in T$ *then* $\downarrow^T t = \downarrow^T t' \in \mathcal{N}f$.

4.7 Computation with Natural Numbers

In this section we show that the eliminations for natural numbers are accurately modeled.

Lemma 4.11 (Case). *If* $a = a' \in \mathsf{Nat}\,(\alpha + 1)$ *and* $B = B' \in \mathsf{Nat}\,(\alpha + 1) \to \mathsf{Set}_\ell$ *and* $f_z = f_z' \in B\,(\mathsf{zero}\langle\beta\rangle)$ *and* $f_s = f_s' \in (x\!:\!\mathsf{Nat}\,\alpha) \to B\,(\mathsf{suc}\langle\gamma\rangle x)$ *then* $a\,\mathsf{case}_\ell\,B\,f_z\,f_s = a'\,\mathsf{case}_\ell\,B'\,f_z'\,f_s' \in B\,a$.

Proof. By induction on $a = a' \in \mathsf{Nat}\,(\alpha + 1)$. □

Lemma 4.12 ($\mathcal{N}at$ is cocontinuous). $\mathcal{N}at(\infty) = \bigcup_{\alpha<\infty} \mathcal{N}at(\alpha)$.

Proof. By induction on $a = a' \in \mathcal{N}at(\infty)$, we can easily show $a = a' \in \mathcal{N}at(\alpha)$ for some $\alpha < \infty$. For instance, α could be the number of uses of the successor rule plus one. □

As the semantic counterpart of judgement $\Gamma \vdash T : \mathsf{Adm}\,\ell$, let us write $\boxed{B = B' \in \mathsf{Adm}\,\ell}$ iff $B = B' \in \mathsf{FixK}\,\ell$ and for all $\beta \in \mathcal{S}ize$ and $a \in \mathsf{Nat}\,\beta$ we have $B\,\beta\,a \le B\,\infty\,a \in \mathcal{T}ype$ and $B'\,\beta\,a \le B'\,\infty\,a \in \mathcal{T}ype$.

Lemma 4.13 (Fix). *Let* $g = a\,\mathsf{fix}_\ell\,B\,f$ *and* $g' = a'\,\mathsf{fix}_\ell\,B'\,f'$. *If* $a = a' \in \mathsf{Nat}\,\alpha$ *and* $B = B' \in \mathsf{Adm}\,\ell$ *and* $f = f' \in \mathsf{FixT}\,B$ *then* $g = g' \in B\,\alpha\,a$.

Proof. By well-founded induction on α. □

4.8 Fundamental Theorem

In this section we show that the declarative judgements are sound, in particular, well-formed syntactic types map to semantic types, and definitionally equal terms map to related values in the PER model. The proof runs the usual course. First, we define inductively a PER of substitutions

$$
\begin{array}{lcl}
\models () & :\Longleftrightarrow & \text{true} \\
\models \Gamma.\star\mathsf{Size} & :\Longleftrightarrow & \models \Gamma \\
\models \Gamma.s & :\Longleftrightarrow & \models \Gamma \\
\models \Gamma.T & :\Longleftrightarrow & \models \Gamma \text{ and } \Gamma^{\oplus} \models T \\
\\
\Gamma \models s & :\Longleftrightarrow & \models \Gamma \\
\Gamma \models T & :\Longleftrightarrow & \Gamma \models T = T \\
\Gamma \models T = T' & :\Longleftrightarrow & \Gamma \models T = T' : s \text{ for some } s \\
\\
\Gamma \models T : \mathsf{Adm}\ \ell & :\Longleftrightarrow & \Gamma \models T : \mathsf{FixK}\,\ell \text{ and } T\eta = T'\eta' \in \mathsf{Adm}\ \ell \text{ for all } \eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma \\
\Gamma \models T \le T' & :\Longleftrightarrow & \Gamma \models T \text{ and } \Gamma \models T' \text{ and } T\eta \le T'\eta' \in \mathcal{T}\mathit{ype} \text{ for all } \eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma \\
\\
\Gamma \models t : T & :\Longleftrightarrow & \Gamma \models t = t : T \\
\Gamma \models t = t' : T & :\Longleftrightarrow & \models \Gamma.T \text{ and } t\eta = t'\eta' \in T\rho \text{ for all } \eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma \\
\\
\Gamma \models \sigma : \Delta & :\Longleftrightarrow & \Gamma \models \sigma = \sigma \mathrel{:\!\!=} \sigma : \Delta \\
\Gamma \models \sigma = \sigma' \mathrel{:\!\!=} \tau : \Delta & :\Longleftrightarrow & \models \Gamma \text{ and } \models \Delta \text{ and } \sigma\eta = \sigma'\eta' \mathrel{:\!\!=} \tau\rho \in \Delta \text{ for all } \eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma
\end{array}
$$

Fig. 9. Semantic judgements.

$\boxed{\eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma}$.

$$
\frac{}{() = () \mathrel{:\!\!=} () \in ()} \qquad \frac{\eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma \qquad T\rho \in \mathcal{S}\mathit{et}_\ell \qquad u = u' = t \in T\rho}{(\eta, u) = (\eta', u') \mathrel{:\!\!=} (\rho, t) \in \Gamma.T}
$$

$$
\frac{\eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma \qquad \alpha \in \mathcal{S}\mathit{ize}}{(\eta, \alpha) = (\eta', \alpha) \mathrel{:\!\!=} (\rho, \alpha) \in \Gamma.\mathsf{Size}} \qquad \frac{\eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma \qquad \alpha, \alpha', \beta \in \mathcal{S}\mathit{ize}}{(\eta, \alpha) = (\eta', \alpha') \mathrel{:\!\!=} (\rho, \beta) \in \Gamma.\dot{-}\mathsf{Size}}
$$

We write $\rho \in \Gamma$ for $\rho = \rho \mathrel{:\!\!=} \rho \in \Gamma$.

Lemma 4.14 (Resurrection). *If* $\eta = \eta' \mathrel{:\!\!=} \rho \in \Gamma$ *then* $\rho \in \Gamma^{\oplus}$.

Then, in Fig. 9, we define semantic counterparts of our declarative judgements by recursion on the length of the context.

Theorem 4.15 (Fundamental theorem).

(1) *If* $\vdash \Gamma$ *then* $\models \Gamma$.
(2) *If* $\Gamma \vdash J$ *then* $\Gamma \models J$.

Proof. Simultaneously, by induction on the derivation. □

4.9 Completeness of NbE

From the fundamental theorem, we harvest completeness of NbE in this section, i. e., we show that definitionally equal terms have the same normal form. We may write simply Γ for its length $|\Gamma|$ when there is no danger of confusion, for instance in de Bruijn level x_Γ or in read back R_Γ. We define the *identity environment* $\boxed{\rho_\Gamma}$ by induction on Γ, setting $\rho_{()} = ()$ and $\rho_{\Gamma.\star\mathsf{Size}} = (\rho_\Gamma, \mathsf{x}_\Gamma)$ and $\rho_{\Gamma.T} = (\rho_\Gamma, \uparrow^{T\rho_\Gamma} \mathsf{x}_\Gamma)$.

Lemma 4.16 (Identity environment). *If* $\vdash \Gamma$ *then* $\rho_\Gamma \in \Gamma$.

We now define the normalization relation $\boxed{\mathrm{nbe}^T_\Gamma t \searrow v} :\Longleftrightarrow \mathrm{R}_\Gamma \downarrow^{T\rho_\Gamma}(t\rho_\Gamma) \searrow v$. Whenever $\mathrm{nbe}^T_\Gamma t \searrow v$, we may write $\mathrm{nbe}^T_\Gamma t$ for v.

Theorem 4.17 (Completeness of NbE). *If $\Gamma \vdash t = t' : T$ then there are normal forms $v \approx v'$ such that $\mathrm{nbe}^T_\Gamma t \searrow v$ and $\mathrm{nbe}^T_\Gamma t' \searrow v'$.*

Proof. By the fundamental theorem, $T\rho_\Gamma \in \mathcal{S}et_\ell$ for some ℓ and $t\rho_\Gamma = t'\rho_\Gamma \in T\rho_\Gamma$. By reification (Cor. 4.10) we have $\downarrow^{T\rho_\Gamma}(t\rho_\Gamma) = \downarrow^{T\rho_\Gamma}(t'\rho_\Gamma) \in \mathcal{N}f$ which implies the theorem by read back with $k = |\Gamma|$. □

5 SOUNDNESS OF NORMALIZATION BY EVALUATION

In this section, we show that NbE is sound for judgmental equality, i.e., that *same normal form* implies *definitional equality*. The proof follows Abel et al. [2007] and Fridlender and Pagano [2013] and defines a Kripke logical relation $\Gamma \vdash t : T \circledR f \in A$ between a well-typed term $\Gamma \vdash t : T$ and a value $f \in A$.

First, let us define some auxiliary judgements that relate a well-formed syntactic object to a value, via read back. They will constitute the logical relation for base types, but need to be strengthened for function types.

$$
\begin{array}{lcl}
\Gamma \vdash a \doteq \mathrm{R}^{\mathsf{size}}\,\alpha & :\Longleftrightarrow & \forall \xi : \Gamma' \le \Gamma.\ \mathrm{R}^{\mathsf{size}}_{\Gamma'}\,\alpha \searrow a\xi \\
\Gamma \vdash T \doteq \mathrm{R}^{\mathsf{ty}}\,A : s & :\Longleftrightarrow & \forall \xi : \Gamma' \le \Gamma.\ \exists V.\ \mathrm{R}^{\mathsf{ty}}_{\Gamma'}\,A \searrow V \ \text{ and } \Gamma' \vdash T\xi = V : s \\
\Gamma \vdash t \doteq \mathrm{R}\,d : T & :\Longleftrightarrow & \forall \xi : \Gamma' \le \Gamma.\ \exists v.\ \mathrm{R}_{\Gamma'}\,d \searrow v \ \text{ and } \Gamma' \vdash t\xi = v : T\xi \\
\Gamma \vdash t \doteq \mathrm{R}^{\mathsf{ne}}\,n : T & :\Longleftrightarrow & \forall \xi : \Gamma' \le \Gamma.\ \exists m.\ \mathrm{R}^{\mathsf{ne}}_{\Gamma'}\,n \searrow m \ \text{ and } \Gamma' \vdash t\xi = m : T\xi
\end{array}
$$

By definition, these relations are closed under subsumption and weakening, e.g., if $\Gamma \vdash t \doteq \mathrm{R}\,d : T$ and $\Gamma \vdash T \le T'$ then $\Gamma \vdash t \doteq \mathrm{R}\,d : T'$, and if $\xi : \Gamma' \le \Gamma$ then $\Gamma' \vdash t\xi \doteq \mathrm{R}\,d : T\xi$.

Lemma 5.1 (Closure properties for neutrals).

(1) *If $\Gamma \vdash t \doteq \mathrm{R}^{\mathsf{ne}}\,n : \Pi\,U\,T$ and $\Gamma \vdash u \doteq \mathrm{R}\,d : U$ then $\Gamma \vdash t\,u \doteq \mathrm{R}^{\mathsf{ne}}\,n\,d : T[u]$.*
(2) *If $\Gamma \vdash t \doteq \mathrm{R}^{\mathsf{ne}}\,n : \Pi\,\mathrm{Size}\,T$ and $\Gamma \vdash a \doteq \mathrm{R}^{\mathsf{size}}\alpha$ then $\Gamma \vdash t\,a \doteq \mathrm{R}^{\mathsf{ne}}\,n\,\alpha : T[a]$.*
(3) *If $\Gamma \vdash t \doteq \mathrm{R}^{\mathsf{ne}}\,n : \forall\,T$ and $\Gamma^{\oplus} \vdash a, b : \mathrm{Size}$ and $\alpha \in \mathcal{S}ize$ then $\Gamma \vdash t\,\langle a\rangle \doteq \mathrm{R}^{\mathsf{ne}}\,n\,\langle\alpha\rangle : T[b]$.*

Let $\boxed{\Gamma \vdash T \searrow W : s}$ denote the conjunction of $T \searrow W$ and $\Gamma \vdash T = W : s$. We simultaneously define $\boxed{\Gamma \vdash T' \circledR A' \in s}$ for $\Gamma \vdash T' : s$ and $\boxed{\Gamma \vdash t : T' \circledR f \in A'}$ for $\Gamma \vdash t : T'$ and $f \in A'$ by induction on $A' \in s$.

Case $A' \searrow N$ neutral.

$$
\begin{array}{lcl}
\Gamma \vdash T' \circledR A' \in s & :\Longleftrightarrow & \Gamma \vdash T' \searrow n : s \text{ for some neutral } n \text{ and } \Gamma \vdash T' \doteq \mathrm{R}^{\mathsf{ty}}\,A' : s. \\
\Gamma \vdash t : T' \circledR f \in A' & :\Longleftrightarrow & \Gamma \vdash t \doteq \mathrm{R}\downarrow^{A'} f : T'.
\end{array}
$$

Case $A' \searrow \mathrm{Nat}\,\alpha$.

$$
\begin{array}{lcl}
\Gamma \vdash T' \circledR A' \in s & :\Longleftrightarrow & \Gamma \vdash T' \searrow \mathrm{Nat}\,a : s \text{ for some } a \text{ and } \Gamma \vdash a \doteq \mathrm{R}^{\mathsf{size}}\alpha. \\
\Gamma \vdash t : T' \circledR f \in A' & :\Longleftrightarrow & \Gamma^{\oplus} \vdash T' \searrow \mathrm{Nat}\,a : s \text{ for some } a \text{ and } \Gamma^{\oplus} \vdash a \doteq \mathrm{R}^{\mathsf{size}}\alpha \\
& & \quad \text{and } \Gamma \vdash t \doteq \mathrm{R}\downarrow^{A'} f : \mathrm{Nat}\,a.
\end{array}
$$

Case $A' \searrow \mathrm{Set}_{\ell'}$.

$$
\begin{array}{lcl}
\Gamma \vdash T' \circledR A' \in s & :\Longleftrightarrow & \Gamma \vdash T' \searrow \mathrm{Set}_{\ell'} : s. \\
\Gamma \vdash U : T' \circledR B \in A' & :\Longleftrightarrow & \Gamma^{\oplus} \vdash T' \searrow \mathrm{Set}_{\ell'} : s \text{ and } \Gamma \vdash U \circledR B \in \mathrm{Set}_{\ell'}.
\end{array}
$$

Case $A' \searrow \Pi\, A\, B$.

$$\Gamma \vdash T' \circledR A' \in s \quad :\Longleftrightarrow \quad \Gamma \vdash T' \searrow \Pi\, U\, T : s \text{ for some } U, T \text{ and } \Gamma \vdash U \circledR A \in s$$
$$\text{and } \forall \xi : \Gamma' \leq \Gamma.\ \Gamma' \vdash u : U\xi \circledR a \in A \implies \Gamma' \vdash T(\xi, u) \circledR B[a] \in s.$$

$$\Gamma \vdash t : T' \circledR f \in A' \quad :\Longleftrightarrow \quad \Gamma^{\oplus} \vdash T' \searrow \Pi\, U\, T : s \text{ for some } U, T \text{ and } \Gamma^{\oplus} \vdash U \circledR A \in s$$
$$\text{and } \forall \xi : \Gamma' \leq \Gamma.\ \Gamma' \vdash u : U\xi \circledR a \in A \implies \Gamma' \vdash t\xi\, u : T(\xi, u) \circledR f\, a \in B[a].$$

Case $A' \searrow \Pi\, \mathrm{Size}\, B$.

$$\Gamma \vdash T' \circledR A' \in s \quad :\Longleftrightarrow \quad \Gamma \vdash T' \searrow \Pi\, \mathrm{Size}\, T : s \text{ for some } T$$
$$\text{and } \forall \xi : \Gamma' \leq \Gamma.\ \Gamma' \vdash a \doteq \mathrm{R}^{\mathrm{size}}\alpha \implies \Gamma' \vdash T(\xi, a) \circledR B[\alpha] \in s.$$

$$\Gamma \vdash t : T' \circledR f \in A' \quad :\Longleftrightarrow \quad \Gamma^{\oplus} \vdash T' \searrow \Pi\, \mathrm{Size}\, T : s \text{ for some } T$$
$$\text{and } \forall \xi : \Gamma' \leq \Gamma.\ \Gamma' \vdash a \doteq \mathrm{R}^{\mathrm{size}}\alpha \implies \Gamma' \vdash t\xi\, a : T(\xi, a) \circledR f\, \alpha \in B[\alpha].$$

Case $A' \searrow \forall\, B$.

$$\Gamma \vdash T' \circledR A' \in s \quad :\Longleftrightarrow \quad \Gamma \vdash T' \searrow \forall\, T : s \text{ for some } T$$
$$\text{and } \forall \xi : \Gamma' \leq \Gamma,\ \Gamma' \vdash b : \mathrm{Size},\ \beta \in \mathcal{S}ize.\ \Gamma' \vdash b \doteq \mathrm{R}^{\mathrm{size}}\beta \implies \Gamma' \vdash T(\xi, b) \circledR B[\beta] \in s.$$

$$\Gamma \vdash t : T' \circledR f \in A' \quad :\Longleftrightarrow \quad \Gamma^{\oplus} \vdash T' \searrow \forall\, T : s \text{ for some } T$$
$$\text{and } \forall \xi : \Gamma' \leq \Gamma,\ \Gamma'^{\oplus} \vdash a, b : \mathrm{Size},\ \alpha, \beta \in \mathcal{S}ize.$$
$$\Gamma'^{\oplus} \vdash b \doteq \mathrm{R}^{\mathrm{size}}\beta \implies \Gamma' \vdash t\xi\, \langle a \rangle : T(\xi, b) \circledR f\, \langle \alpha \rangle \in B[\beta].$$

We may prove theorems "by induction on $\Gamma \vdash T \circledR A \in s$", even if in reality this will be proofs by induction on $A \in s$ and cases on $\Gamma \vdash T \circledR A \in s$. We write $\boxed{\Gamma \vdash T \circledR A}$ if $\Gamma \vdash T \circledR A \in s$ for some sort s. The logical relations are closed under weakening.

Theorem 5.2 (Into and out of the logical relation). *Let* $\Gamma \vdash T \circledR A \in s$ *and* $A \sqsubseteq S$. *Then:*
(1) *If* $\Gamma \vdash t \doteq \mathrm{R}^{\mathrm{ne}}\, n : T$ *then* $\Gamma \vdash t : T \circledR \uparrow^S n \in A$.
(2) *If* $\Gamma \vdash t : T \circledR f \in A$ *then* $\Gamma \vdash t \doteq \mathrm{R}\downarrow^S f : T$.
(3) $\Gamma \vdash T \doteq \mathrm{R}^{\mathrm{ty}}\, A : s$.

Proof. Simultaneously by induction on $\Gamma \vdash T \circledR A \in s$. □

Lemma 5.3 (Semantic implies judgmental subtyping [Fridlender and Pagano 2013]).
(1) *If* $\Gamma \vdash a \doteq \mathrm{R}^{\mathrm{size}}\alpha$ *and* $\Gamma \vdash b \doteq \mathrm{R}^{\mathrm{size}}\beta$ *and* $\alpha \leq \beta$ *then* $a \leq b$.
(2) *If* $\Gamma \vdash T \circledR A$ *and* $\Gamma \vdash T' \circledR A'$ *and* $A \leq A' \in \mathcal{T}ype$ *then* $\Gamma \vdash T \leq T'$.

Lemma 5.4 (Subsumption for the logical relation [Fridlender and Pagano 2013]). *If* $\Gamma \vdash T \circledR A$ *and* $\Gamma \vdash T' \circledR A'$ *and* $A \leq A' \in \mathcal{T}ype$ *then* $\Gamma \vdash t : T \circledR f \in A$ *implies* $\Gamma \vdash t : T' \circledR f \in A'$.

Fig. 10 defines a logical relation for substitutions $\boxed{\Gamma \vdash \sigma \coloneqq \tau : \Delta \circledR \eta \coloneqq \rho}$. We write $\boxed{\Gamma \vdash \tau : \Delta \circledR \rho}$ for $\Gamma \vdash \tau \coloneqq \tau : \Delta \circledR \rho \coloneqq \rho$.

The following judgements are used to state the fundamental theorem of typing.

$$\Gamma \Vdash t : T \quad :\Longleftrightarrow \quad \Gamma' \vdash t\sigma : T\tau \circledR t\eta \in T\rho \text{ for all } \Gamma' \vdash \sigma \coloneqq \tau : \Gamma \circledR \eta \coloneqq \rho$$

$$\Gamma \Vdash \sigma_0 : \Delta \quad :\Longleftrightarrow \quad \Gamma' \vdash \sigma_0\sigma \coloneqq \sigma_0\tau : \Delta \circledR \sigma_0\eta \coloneqq \sigma_0\rho \text{ for all } \Gamma' \vdash \sigma \coloneqq \tau : \Gamma \circledR \eta \coloneqq \rho$$

Theorem 5.5 (Fundamental theorem of typing).
(1) *If* $\Gamma \vdash t : T$ *then* $\Gamma \Vdash t : T$.
(2) *If* $\Gamma \vdash \sigma : \Delta$ *then* $\Gamma \Vdash \sigma : \Delta$.

Proof. Each by induction on the derivation. □

Lemma 5.6 (Identity environment). *If* $\vdash \Gamma$ *then* $\Gamma \vdash \mathrm{id} : \Gamma \circledR \rho_\Gamma$.

$$\frac{\vdash \Gamma}{\Gamma \vdash () \coloneqq () : () \circledR () \coloneqq ()} \qquad \frac{\Gamma \vdash \sigma \coloneqq \tau : \Delta \circledR \eta \coloneqq \rho \qquad \Gamma \vdash a : \mathsf{Size} \qquad \Gamma \vdash a \doteq \mathsf{R}^{\mathsf{size}}\alpha}{\Gamma \vdash (\sigma, a) \coloneqq (\tau, a) : \Delta.\mathsf{Size} \circledR (\eta, \alpha) \coloneqq (\rho, \alpha)}$$

$$\frac{\Gamma \vdash \sigma \coloneqq \tau : \Delta \circledR \eta \coloneqq \rho \qquad \Gamma^{\oplus} \vdash a, b : \mathsf{Size} \qquad \alpha, \beta \in \mathcal{S}ize \qquad \Gamma^{\oplus} \vdash b \doteq \mathsf{R}^{\mathsf{size}}\beta}{\Gamma \vdash (\sigma, a) \coloneqq (\tau, b) : \Delta.^{\div}\mathsf{Size} \circledR (\eta, \alpha) \coloneqq (\rho, \beta)}$$

$$\frac{\Gamma \vdash \sigma \coloneqq \tau : \Delta \circledR \eta \coloneqq \rho \qquad \Delta^{\oplus} \vdash T \qquad \Gamma \vdash u = t : T\tau \qquad \Gamma \vdash t : T\tau \circledR f \in T\rho \qquad f = g \in T\rho}{\Gamma \vdash (\sigma, u) \coloneqq (\tau, t) : \Delta.T \circledR (\eta, f) \coloneqq (\rho, q)}$$

Fig. 10. Logical relation for substitutions $\boxed{\Gamma \vdash \sigma \coloneqq \tau : \Delta \circledR \eta \coloneqq \rho}$.

Corollary 5.7 (Soundness of NbE).
(1) *If* $\Gamma \vdash t : T$ *then* $\Gamma \vdash t = \mathsf{nbe}^T_\Gamma t : T$.
(2) *If* $\Gamma \vdash t, t' : T$ *and* $\mathsf{nbe}^T_\Gamma t \approx \mathsf{nbe}^T_\Gamma t'$ *then* $\Gamma \vdash t = t' : T$.

Proof. (1) For the identity environment $\Gamma \vdash \mathsf{id} : \Gamma \circledR \rho_\Gamma$ (Lemma 5.6) the Fundamental Theorem for Typing gives $\Gamma \vdash t : T \circledR t\rho_\Gamma \in T\rho_\Gamma$. This implies $\mathsf{R}_\Gamma \downarrow^{(T\rho_\Gamma)}(t\rho_\Gamma) \searrow v$ for some normal form v and $\Gamma \vdash t = v : T$ by Thm. 5.2. Then (2): From (1), using Lemma 3.1: $\Gamma \vdash t = \mathsf{nbe}^T_\Gamma t = \mathsf{nbe}^T_\Gamma t' = t' : T$. □

Corollary 5.8 (Decidability of judgemental equality). *If* $\Gamma \vdash t, t' : T$ *then the test whether* $\mathsf{nbe}^T_\Gamma t \approx \mathsf{nbe}^T_\Gamma t'$ *terminates and decides* $\Gamma \vdash t = t' : T$.

From correctness of NbE and the logical relations we can further prove injectivity of type constructors, inversion of subtyping, and subject reduction. The proofs follow roughly Fridlender and Pagano [2013], for details, see the long version of this article.

6 ALGORITHMIC SUBTYPING

Fig 11 defines an incremental subtyping algorithm $\boxed{\Gamma \vdash T <: T'}$. Neutral types are subtypes iff they are equal, which is checked using NbE.

$$\frac{T \searrow n \qquad T' \searrow n' \qquad \mathsf{Nbe}_\Gamma n \approx \mathsf{Nbe}_\Gamma n'}{\Gamma \vdash T <: T'} \qquad \frac{T \searrow \mathsf{Set}_\ell \qquad T' \searrow \mathsf{Set}_{\ell'}}{\Gamma \vdash T <: T'}\ \ell \le \ell'$$

$$\frac{T \searrow \mathsf{Nat}\, a \qquad T' \searrow \mathsf{Nat}\, a'}{\Gamma \vdash T <: T'}\ a \le a' \qquad \frac{T_1' \searrow \Pi^\star \mathsf{Size}\, T_1 \qquad T_2' \searrow \Pi^\star \mathsf{Size}\, T_2 \qquad \Gamma.\mathsf{Size} \vdash T_1 <: T_2}{\Gamma \vdash T_1' <: T_2'}$$

$$\frac{T_1' \searrow \Pi\, U_1\, T_1 \qquad T_2' \searrow \Pi\, U_2\, T_2 \qquad \Gamma \vdash U_2 <: U_1 \qquad \Gamma.U_2 \vdash T_1 <: T_2}{\Gamma \vdash T_1' <: T_2'}$$

Fig. 11. Algorithmic subtyping $\boxed{\Gamma \vdash T <: T'}$.

Lemma 6.1 (Soundness of algorithmic subtyping). *If* $\Gamma \vdash T <: T'$ *then* $\Gamma \vdash T \le T'$.

Proof. By induction on $\Gamma \vdash T <: T'$, soundness of NbE, and subject reduction. □

Lemma 6.2 (Semantic subtyping implies algorithmic subtyping).
If $\Gamma \vdash T \circledR A$ *and* $\Gamma \vdash T' \circledR A'$ *and* $A \leq A' \in \mathcal{T}ype$ *then* $\Gamma \vdash T <: T'$.

Proof. By induction on $\Gamma \vdash T \circledR A$ and $\Gamma \vdash T' \circledR A'$ and cases on $A \leq A' \in \mathcal{T}ype$. □

Corollary 6.3 (Completeness of algorithmic subtyping). *If* $\Gamma \vdash T \leq T'$ *then* $\Gamma \vdash T <: T'$.

Proof. By the fundamental theorems $\Gamma \vdash T \circledR T\rho_\Gamma$ and $\Gamma \vdash T' \circledR T'\rho_\Gamma$ and $T\rho_\Gamma \leq T'\rho_\Gamma \in \mathcal{T}ype$. By Lemma 6.2, $\Gamma \vdash T <: T'$. □

Lemma 6.4 (Termination of algorithmic subtyping). *If* $\Gamma \vdash T \circledR A$ *and* $\Gamma \vdash T' \circledR A'$ *then the query* $\Gamma \vdash T <: T'$ *terminates.*

Proof. By induction on $A \in s$ and $A' \in s'$ and cases on $\Gamma \vdash T \circledR A$ and $\Gamma \vdash T' \circledR A'$. □

Theorem 6.5 (Decidability of subtyping). *If* $\Gamma \vdash T, T'$, *then* $\Gamma \vdash T \leq T'$ *is decided by the query* $\Gamma \vdash T <: T'$.

Proof. By the fundamental theorem of typing, $\Gamma \vdash T \circledR A$ and $\Gamma \vdash T' \circledR A'$, thus, the query $\Gamma \vdash T <: T'$ terminates by Lemma 6.4. If successfully, then $\Gamma \vdash T \leq T'$ by soundness of algorithmic equality. Otherwise $\Gamma \vdash T \leq T'$ is impossible by completeness of algorithmic equality. □

7 TYPE CHECKING

In this section, we show that type checking for normal forms is decidable, and succeeds for those which can be typed via the restricted rule for size polymorphism elimination:

$$\frac{\Gamma \vdash_s t : \forall T \qquad \Gamma^\oplus \vdash a : \mathsf{Size}}{\Gamma \vdash_s t\,\langle a\rangle : T[a]}$$

We refer to the restricted typing judgement as $\boxed{\Gamma \vdash_s t : T}$, and obviously, if $\Gamma \vdash_s t : T$ then $\Gamma \vdash t : T$.

Figure 12 displays the rules for bidirectional typing of normal forms. Note that we could go beyond normal forms, by adding inference rules for the Nat -constructors:

$$\frac{\Gamma^\oplus \vdash a : \mathsf{Size}}{\Gamma \vdash \mathsf{zero}\langle a\rangle \rightrightarrows \mathsf{Nat}\,(a+1)} \qquad \frac{\Gamma^\oplus \vdash a : \mathsf{Size} \qquad \Gamma \vdash t \leftleftarrows \mathsf{Nat}\,a}{\Gamma \vdash \mathsf{suc}\langle a\rangle t \rightrightarrows \mathsf{Nat}\,(a+1)}$$

Theorem 7.1 (Soundness of type checking). *Let* $\vdash \Gamma$.
(1) *If* $\Gamma^\oplus \vdash T$ *and* $\mathcal{D} :: \Gamma \vdash t \leftleftarrows T$ *then* $\Gamma \vdash_s t : T$.
(2) *If* $\mathcal{D} :: \Gamma \vdash t \rightrightarrows T$ *then* $\Gamma^\oplus \vdash T$ *and* $\Gamma \vdash_s t : T$.

Lemma 7.2 (Weak head reduction of subtypes). *Let* $\mathcal{D} :: \Gamma \vdash T <: T'$.
(1) *If* $T' \searrow \mathsf{Nat}\,a'$ *then* $T \searrow \mathsf{Nat}\,a$ *and* $\Gamma \vdash a <: a' : \mathsf{Size}$.
(2) *If* $T' \searrow \mathsf{Set}_{\ell'}$ *then* $T \searrow \mathsf{Set}_\ell$ *and* $\ell <: \ell'$.
(3) *If* $T' \searrow \Pi\,A'\,B'$ *then* $T \searrow \Pi\,A\,B$ *and* $\Gamma \vdash A' <: A$ *and* $\Gamma.A' \vdash B <: B'$.
(4) *If* $T' \searrow \Pi^\star \mathsf{Size}\,B'$ *and* $T \searrow \Pi^\star \mathsf{Size}\,B$ *and* $\Gamma.\mathsf{Size} \vdash B <: B'$.

Proof. By cases on $\mathcal{D}$, since weak head evaluation is deterministic. □

This lemma also holds in the other direction of subtyping, i. e., when $T <: T'$ and T weak head evaluates, then T' weak head evaluates to a type of the same form.

Lemma 7.3 (Subsumption for type checking). *Let* $\mathsf{id} : \Gamma' \leq \Gamma$.
(1) *If* $\mathcal{D} :: \Gamma \vdash t \leftleftarrows T$ *and* $\Gamma^\oplus \vdash T \leq T'$ *then* $\Gamma' \vdash t \leftleftarrows T'$.
(2) *If* $\mathcal{D} :: \Gamma \vdash t \rightrightarrows T$ *then* $\Gamma' \vdash t \rightrightarrows T'$ *and* $\Gamma'^\oplus \vdash T \leq T'$.

Checking $\boxed{\Gamma \vdash t \leftleftarrows T}$. Input: Γ, t, T. Output: *yes/no*.

$$\frac{T' \searrow s \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash \mathsf{Nat}\, a \leftleftarrows T'} \qquad \frac{T' \searrow \mathsf{Set}_{\ell'} \qquad \ell < \ell'}{\Gamma \vdash \mathsf{Set}_{\ell} \leftleftarrows T'}$$

$$\frac{T' \searrow s \qquad \Gamma \vdash U \leftleftarrows s \qquad \Gamma.U \vdash T \leftleftarrows s}{\Gamma \vdash \Pi\, U\, T \leftleftarrows T'} \qquad \frac{T' \searrow s \qquad \Gamma.\mathsf{Size} \vdash T \leftleftarrows s}{\Gamma \vdash \Pi^{\star}\mathsf{Size}\, T \leftleftarrows T'}$$

$$\frac{T' \searrow \mathsf{Nat}\, b \qquad \Gamma^{\oplus} \vdash a + 1 \leq b : \mathsf{Size}}{\Gamma \vdash \mathsf{zero}\langle a\rangle \leftleftarrows T'} \qquad \frac{T' \searrow \mathsf{Nat}\, b \qquad \Gamma^{\oplus} \vdash a + 1 \leq b : \mathsf{Size} \qquad \Gamma \vdash t \leftleftarrows \mathsf{Nat}\, a}{\Gamma \vdash \mathsf{suc}\langle a\rangle t \leftleftarrows T'}$$

$$\frac{T' \searrow \Pi^{\star} U\, T \qquad \Gamma.^{\star}U \vdash t \leftleftarrows T}{\Gamma \vdash \lambda t \leftleftarrows T'} \qquad \frac{\Gamma \vdash t \rightrightarrows T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash t \leftleftarrows T'}$$

Inference $\boxed{\Gamma \vdash t \rightrightarrows T}$. Input: Γ, t. Output: T or *no*.

$$\frac{\Gamma(i) = {}^{\cdot}T}{\Gamma \vdash \mathsf{v}_i \rightrightarrows T} \qquad \frac{\Gamma \vdash t \rightrightarrows T' \qquad T' \searrow \Pi\, U\, T \qquad \Gamma \vdash u \leftleftarrows U}{\Gamma \vdash t\, u \rightrightarrows T[u]}$$

$$\frac{\Gamma \vdash t \rightrightarrows T' \qquad T' \searrow \Pi\, \mathsf{Size}\, T \qquad \Gamma \vdash a : \mathsf{Size}}{\Gamma \vdash t\, a \rightrightarrows T[a]} \qquad \frac{\Gamma \vdash t \rightrightarrows T' \qquad T' \searrow \Pi^{\div}\mathsf{Size}\, T \qquad \Gamma^{\oplus} \vdash a : \mathsf{Size}}{\Gamma \vdash t\, \langle a\rangle \rightrightarrows T[a]}$$

$$\frac{\Gamma^{\oplus} \vdash T \leftleftarrows \mathsf{Nat}\,(a+1) \to \mathsf{Set}_{\ell} \qquad \begin{array}{c}\Gamma \vdash u \rightrightarrows \mathsf{Nat}\,(a+1)\\ \Gamma \vdash t_z \leftleftarrows T\,(\mathsf{zero}\langle a\rangle)\end{array} \qquad \Gamma \vdash t_s \leftleftarrows (x{:}\mathsf{Nat}\, a) \to T\,(\mathsf{suc}\langle a\rangle x)}{\Gamma \vdash u\, \mathsf{case}_{\ell}\, T\, t_z\, t_s \rightrightarrows T\, u}$$

$$\frac{\Gamma \vdash u \rightrightarrows \mathsf{Nat}\, a \qquad \Gamma^{\oplus} \vdash T \leftleftarrows \mathsf{FixK}\, \ell \qquad \Gamma \vdash t \leftleftarrows \mathsf{FixT}\, T}{\Gamma \vdash u\, \mathsf{fix}_{\ell}\, T\, t \rightrightarrows T\, a\, u}$$

Fig. 12. Bidirectional type-checking of normal forms.

Proof. Simultaneously by induction on $\mathscr{D}$, using lemma 7.2 and soundness and completeness of algorithmic subtyping. □

Theorem 7.4 (Completeness of type checking for normal terms).
(1) *If* $\mathscr{D} :: \Gamma \vdash_{\mathsf{s}} v : T$ *then* $\Gamma \vdash v \leftleftarrows T$.
(2) *If* $\mathscr{D} :: \Gamma \vdash_{\mathsf{s}} m : T$ *then* $\Gamma \vdash m \rightrightarrows U$ *and* $\Gamma^{\oplus} \vdash U \leq T$.

Proof. Simultaneously by induction on $\mathscr{D}$, using (strong) inversion and Lemma 7.3. □

Lemma 7.5 (Termination of type checking). *Let* $\vdash \Gamma$.
(1) *The query* $\Gamma \vdash t \rightrightarrows ?$ *terminates.*
(2) *If* $\Gamma^{\oplus} \vdash T$ *then the query* $\Gamma \vdash t \leftleftarrows T$ *terminates.*

Proof. By induction on t, using type weak head normalization and soundness of type checking, to maintain well-formedness of types. And, of course, decidability of subtyping. □

Theorem 7.6 (Decidability of type checking for normal terms). *Let* $\vdash \Gamma$ *and* $\Gamma^{\oplus} \vdash T$. *Then* $\Gamma \vdash_{\mathsf{s}} v : T$ *is decided by* $\Gamma \vdash v \leftleftarrows T$.

8 DISCUSSION AND CONCLUSIONS

In this article, we have described the first successful integration of higher-rank size polymorphism into a core type theory with dependent function types, a sized type of natural numbers, a predicative hierarchy of universes, subtyping, and η-equality. This is an important stepping stone for the smooth integration of sized types into dependently-typed proof assistants. In these final paragraphs, we discuss some questions and insights that follow from our work and go beyond it.

It is now straightforward to add a unit type $\mathbf{1}$ with extensional equality $t = * : \mathbf{1}$ for all $t : \mathbf{1}$. We simply extend reification such that $\downarrow^{\mathbf{1}} a = *$. Further, $\mathbf{1}$ is a new type shape with rule $\mathbf{1} \sqsubseteq \mathbf{1}$.

In the long run, we wish for a type-directed equality check that does not do normalization in one go, but interleaves weak head normalization with structural comparison. Such an equality test is at the heart of Agda's type checker and it generates constraints for meta variables involved in type reconstruction [Norell 2007]. However, the usual bidirectional construction [Abel and Scherer 2012] does not seem to go through as we lack uniqueness of types (and even principal types).

For now, we have only exploited shape-irrelevance of sized types, but this directly extends to universe levels. If we consider all universes as a single shape $\mathrm{Set}_{\ell_1} \sqsubseteq \mathrm{Set}_{\ell_2}$, we can quantify over levels irrelevantly, as Set is a shape-irrelevant type constructor. This is a stepping stone for integrating universe cumulativity with Agda's explicit universe-polymorphism. If levels are no longer unique (because of subsumption), they will get in the way of proofs, analogously to sizes. With an irrelevant quantifier we can ignore levels where they do not matter. We will still respect them where they matter, thus, we keep consistency.

Our reflections on level irrelevance lead us to the question: can a type theory T with a stratified universe hierarchy be understood as a sort of refinement of the inconsistent System U (Type:Type)? Intuitively, when checking two terms of T for equality, could we ignore the stratification in the type A which directs the equality check (thus, consider A coming from U)? Such a perspective would put stratification in one pot with size assignment: Size annotations and levels are both just annotations for the termination checker, but do not bear semantic relevance. We could switch the universe checker temporarily off as we do with the termination checker—cf. the work of Stump et al. [2010] on *termination casts*.

Finally, we would like a general theory of shape-irrelevance that extends beyond size-indexed types. For instance, any data type constructor could be considered shape-irrelevant in all its indices, with the consequence that index arguments in the data constructors could be declared irrelevant. However, our notion of judgmental equality does not support irrelevant arguments of dependent type. It works for the non-dependent type Size, but we also relied on having a closed inhabitant ∞ in Size. More research is needed to tell a more general story of shape-irrelevance.

ACKNOWLEDGMENTS

This material is based upon work supported by the Swedish Research Council (Vetenskapsrådet) under Grant No. 621-2014-4864 *Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types*. The first author is grateful for recent discussions with Thierry Coquand, Nils Anders Danielsson, and Sandro Stucki which helped clarifying the thoughts leading to this work. He also acknowledges past discussions with Christoph-Simon Senjak. The incentive to write this article came during the EU Cost Action CA15123 EUTYPES meeting in Ljubljana in January 2017; thanks to Andrej Bauer for organizing it.

REFERENCES

Andreas Abel. 2008. Semi-continuous Sized Types and Termination. *Logical Methods in Computer Science* 4, 2:3 (2008), 1–33. https://doi.org/10.2168/LMCS-4(2:3)2008

Andreas Abel. 2010. Towards Normalization by Evaluation for the $\beta\eta$-Calculus of Constructions. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings (Lecture Notes in Computer Science)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.), Vol. 6009. Springer, 224–239. https://doi.org/10.1007/978-3-642-12251-4_17

Andreas Abel. 2012. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. In *Proceedings of the 8th Workshop on Fixed Points in Computer Science (FICS 2012) (Electronic Proceedings in Theoretical Computer Science)*, Dale Miller and Zoltán Ésik (Eds.), Vol. 77. 1–11. http://dx.doi.org/10.4204/EPTCS.77.1

Andreas Abel. 2013. *Normalization by Evaluation: Dependent Types and Impredicativity.* Unpublished. http://www.tcs.ifi.lmu.de/~abel/habil.pdf

Andreas Abel and Thorsten Altenkirch. 2002. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming* 12, 1 (2002), 1–41. https://doi.org/10.1017/S0956796801004191

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2007. Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society Press, 3–12. https://doi.org/10.1109/LICS.2007.33

Andreas Abel, Thierry Coquand, and Miguel Pagano. 2009. A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009, Proceedings (Lecture Notes in Computer Science)*, Pierre-Louis Curien (Ed.), Vol. 5608. Springer, 5–19. https://doi.org/10.1007/978-3-642-02273-9_3

Andreas Abel, Thierry Coquand, and Miguel Pagano. 2011. A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance. *Logical Methods in Computer Science* 7, 2:4 (2011), 1–57. https://doi.org/10.2168/LMCS-7(2:4)2011

Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* 26 (2016), 61. https://doi.org/10.1017/S0956796816000022

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1:29 (2012), 1–36. https://doi.org/10.2168/LMCS-8(1:29)2012

AgdaTeam. 2017. The Agda Wiki. (2017). http://wiki.portal.chalmers.se/agda

Roberto M. Amadio (Ed.). 2008. *Foundations of Software Science and Computational Structures, 11th International Conference, FoSSaCS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*. Lecture Notes in Computer Science, Vol. 4962. Springer. https://doi.org/10.1007/978-3-540-78499-9

Roberto M. Amadio and Solange Coupet-Grimal. 1998. Analysis of a Guard Condition in Type Theory (Extended Abstract).. In *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science)*, Maurice Nivat (Ed.), Vol. 1378. Springer, 48–62. https://doi.org/10.1007/BFb0053541

Henk Barendregt. 1991. Introduction to Generalized Type Systems. *Journal of Functional Programming* 1, 2 (1991), 125–154.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types, See [Amadio 2008], 365–379. https://doi.org/10.1007/978-3-540-78499-9_26

Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. 2004. Type-Based Termination of Recursive Definitions. *Mathematical Structures in Computer Science* 14, 1 (2004), 97–141. https://doi.org/10.1017/S0960129503004122

Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. 2006. CIC$^{\wedge}$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings (Lecture Notes in Computer Science)*, Miki Hermann and Andrei Voronkov (Eds.), Vol. 4246. Springer, 257–271. https://doi.org/10.1007/11916277_18

Gilles Barthe, Benjamin Grégoire, and Colin Riba. 2008a. A Tutorial on Type-Based Termination. In *LerNet ALFA Summer School (Lecture Notes in Computer Science)*, Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto (Eds.), Vol. 5520. Springer, 100–152. https://doi.org/10.1007/978-3-642-03153-3_3

Gilles Barthe, Benjamin Grégoire, and Colin Riba. 2008b. Type-Based Termination with Sized Products. In *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings (Lecture Notes in Computer Science)*, Michael Kaminski and Simone Martini (Eds.), Vol. 5213. Springer, 493–507. https://doi.org/10.1007/978-3-540-87531-4_35

Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse to the Evaluation Functional for Typed λ-calculus. In *Sixth Annual Symposium on Logic in Computer Science (LICS '91), July, 1991, Amsterdam, The Netherlands, Proceedings*. IEEE Computer Society Press, 203–211. https://doi.org/10.1109/LICS.1991.151645

Frédéric Blanqui. 2004. A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004,*

Proceedings (Lecture Notes in Computer Science), Vincent van Oostrom (Ed.), Vol. 3091. Springer, 24–39. https://doi.org/10.1007/978-3-540-25979-4_2

Frédéric Blanqui. 2005. Decidability of Type-Checking in the Calculus of Algebraic Constructions with Size Annotations.. In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings (Lecture Notes in Computer Science)*, C.-H. Luke Ong (Ed.), Vol. 3634. Springer, 135–150. https://doi.org/10.1007/11538363_11

Frédéric Blanqui and Colin Riba. 2006. Combining Typing and Size Constraints for Checking the Termination of Higher-Order Conditional Rewrite Systems. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings (Lecture Notes in Computer Science)*, Miki Hermann and Andrei Voronkov (Eds.), Vol. 4246. Springer, 105–119. https://doi.org/10.1007/11916277_8

Ana Bove. 2009. Another Look at Function Domains. *Electronic Notes in Theoretical Computer Science* 249 (2009), 61–74. https://doi.org/10.1016/j.entcs.2009.07.084

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708. https://doi.org/10.1017/S0960129505004822

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types, In Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction (July 17–21, 1995, Kloster Irsee, Germany). *Science of Computer Programming* 26, 1-3, 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

Olivier Danvy. 1999. Type-Directed Partial Evaluation. In *Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998 (Lecture Notes in Computer Science)*, John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann (Eds.), Vol. 1706. Springer, 367–411. https://doi.org/10.1007/3-540-47018-2_16

N. G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 34 (1972), 381–392.

Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *The Journal of Symbolic Logic* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

Peter Dybjer, Bengt Nordström, and Jan M. Smith (Eds.). 1995. *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*. Lecture Notes in Computer Science, Vol. 996. Springer. https://doi.org/10.1007/3-540-60579-7

Daniel Fridlender and Miguel Pagano. 2013. A Type-Checking Algorithm for Martin-Löf Type Theory with Subtyping Based on Normalisation by Evaluation. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Masahito Hasegawa (Ed.), Vol. 7941. Springer, 140–155. https://doi.org/10.1007/978-3-642-38946-7_12

Herman Geuvers. 1994. A short and flexible proof of Strong Normalization for the Calculus of Constructions, See [Dybjer et al. 1995], 14–38. https://doi.org/10.1007/3-540-60579-7_2

Eduardo Giménez. 1995. Codifying Guarded Definitions with Recursive Schemes, See [Dybjer et al. 1995], 39–59. https://doi.org/10.1007/3-540-60579-7_3

Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002 (SIGPLAN Notices)*, Vol. 37. ACM Press, 235–246. https://doi.org/10.1145/581478.581501

Benjamin Grégoire and Jorge Luis Sacchini. 2010. On Strong Normalization of the Calculus of Constructions with Type-Based Termination. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings (Lecture Notes in Computer Science)*, Christian G. Fermüller and Andrei Voronkov (Eds.), Vol. 6397. Springer, 333–347. https://doi.org/10.1007/978-3-642-16242-8_24

Robert Harper and Frank Pfenning. 2005. On Equivalence and Canonical Forms in the LF Type Theory. *ACM Transactions on Computational Logic* 6, 1 (2005), 61–101. https://doi.org/10.1145/1042038.1042041

Gérard P. Huet. 1989. The Constructive Engine. In *A Perspective in Theoretical Computer Science - Commemorative Volume for Gift Siromoney*, R. Narasimhan (Ed.). World Scientific Series in Computer Science, Vol. 16. World Scientific, 38–69. https://doi.org/10.1142/9789814368452_0004

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 410–423. https://doi.org/10.1145/237721.240882

INRIA. 2016. *The Coq Proof Assistant Reference Manual* (version 8.6 ed.). INRIA. http://coq.inria.fr/

Ugo Dal Lago and Charles Grellois. 2017. Probabilistic Termination by Monadic Affine Sized Typing. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint*

Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science), Hongseok Yang (Ed.), Vol. 10201. Springer, 393–419. https://doi.org/10.1007/978-3-662-54434-1_15

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-Change Principle for Program Termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM Press, 81–92. https://doi.org/10.1145/360204.360210

William Lovas and Frank Pfenning. 2010. Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance. *Logical Methods in Computer Science* 6, 4 (2010). https://doi.org/10.2168/LMCS-6(4:5)2010

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson (Eds.). North-Holland, 73–118.

Alexandre Miquel. 2000. A Model for Impredicative Type Systems, Universes, Intersection Types and Subtyping. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS 2000), 26-29 June 2000, Santa Barbara, California, USA, Proceedings*. IEEE Computer Society Press, 18–29. https://doi.org/10.1109/LICS.2000.855752

Alexandre Miquel. 2001. The Implicit Calculus of Constructions. In *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings (Lecture Notes in Computer Science)*, Samson Abramsky (Ed.), Vol. 2044. Springer, 344–359. https://doi.org/10.1007/3-540-45413-6_27

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems, See [Amadio 2008], 350–364. https://doi.org/10.1007/978-3-540-78499-9

Bengt Nordström. 1988. Terminating General Recursion. *BIT* 28, 3 (1988), 605–619.

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *16th IEEE Symposium on Logic in Computer Science (LICS 2001), 16-19 June 2001, Boston University, USA, Proceedings*. IEEE Computer Society Press, 221–230. https://doi.org/10.1109/LICS.2001.932499

Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society Press, 233–242. https://doi.org/10.1109/LICS.2013.29

Jorge Luis Sacchini. 2014. Linear Sized Types in the Calculus of Constructions. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 169–185. https://doi.org/10.1007/978-3-319-07151-0_11

Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. 2010. Termination Casts: A Flexible Approach to Termination with General Recursion. In *Workshop on Partiality And Recursion in Interactive Theorem Provers, PAR 2010, Satellite Workshop of ITP'10 at FLoC 2010 (Electronic Proceedings in Theoretical Computer Science)*, Ana Bove, Ekaterina Komendantskaya, and Milad Niqui (Eds.), Vol. 43. 76–93. https://doi.org/10.4204/EPTCS.43.6

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, François Pottier and George C. Necula (Eds.). ACM Press, 53–66. https://doi.org/10.1145/1190315.1190324

David Wahlstedt. 2007. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. Ph.D. Dissertation. Chalmers University of Technology.

Benjamin Werner. 1992. A Normalization Proof for an Impredicative Type System with Large Eliminations over Integers. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, Bengt Nordström, Kent Petersson, and Gordon Plotkin (Eds.). 341–357. http://www.cs.chalmers.se/Cs/Research/Logic/Types/proc92.ps

Hongwei Xi. 2002. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation* 15, 1 (2002), 91–131. https://doi.org/10.1023/A:1019916231463

Part III

Conversion

Chapter 7

Decidability of Conversion for Type Theory in Type Theory

Decidability of Conversion for Type Theory in Type Theory

ANDREAS ABEL, Gothenburg University, Sweden
JOAKIM ÖHMAN, IMDEA Software Institute, Spain
ANDREA VEZZOSI, Chalmers University of Technology, Sweden

Type theory should be able to handle its own meta-theory, both to justify its foundational claims and to obtain a verified implementation. At the core of a type checker for intensional type theory lies an algorithm to check equality of types, or in other words, to check whether two types are convertible. We have formalized in Agda a practical conversion checking algorithm for a dependent type theory with one universe à la Russell, natural numbers, and η-equality for Π types. We prove the algorithm correct via a Kripke logical relation parameterized by a suitable notion of equivalence of terms. We then instantiate the parameterized fundamental lemma twice: once to obtain canonicity and injectivity of type formers, and once again to prove the completeness of the algorithm. Our proof relies on inductive-recursive definitions, but not on the uniqueness of identity proofs. Thus, it is valid in variants of intensional Martin-Löf Type Theory as long as they support induction-recursion, for instance, Extensional, Observational, or Homotopy Type Theory.

CCS Concepts: • **Theory of computation → Type theory**; *Proof theory*;

Additional Key Words and Phrases: Dependent types, Logical relations, Formalization, Agda

ACM Reference Format:
Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (January 2018), 29 pages. https://doi.org/10.1145/3158111

1 INTRODUCTION

A fundamental component of the implementation of a typed functional programming language is an algorithm that checks equality of types; even more so for dependently-typed languages where equality of types is non-trivial, as it depends on the equality of terms. In this paper, we consider a dependent type theory with one universe à la Russell, natural numbers, and η-equality for Π types. The algorithm we implement follows the structure of the one used by the dependently-typed language Agda [2017], and has been discussed and refined before in the literature [Abel et al. 2016; Abel and Scherer 2012; Coquand 1991; Harper and Pfenning 2005]. In short, the algorithm will reduce the types or terms under comparison to weak head normal form, compare the heads, and, if they match, recurse on the bodies. Additionally, when comparing terms there is an extra type-directed phase which takes care of η-equality: at function type, we apply the terms under comparison to a fresh variable and continue comparing the results. The type-directed phase could be easily extended to η-equality for other types, by comparing how their elements behave under their eliminators, like it is done in Agda for records, singleton types and others. The proof that the

Authors' addresses: Andreas Abel, Department of Computer Science and Engineering, Gothenburg University, Rännvägen 6b, Göteborg, 41296, Sweden, andreas.abel@gu.se; Joakim Öhman, IMDEA Software Institute, Campus Montegancedo s/n, Madrid, 28223, Spain, joakim.ohman@imdea.org; Andrea Vezzosi, Department of Computer Science and Engineering, Chalmers University of Technology, Rännvägen 6b, Göteborg, 41296, Sweden, vezzosi@chalmers.se.

2475-1421/2018/1-ART23
https://doi.org/10.1145/3158111

algorithm correctly implements equality of types is based on a Kripke logical relation and follows Abel and Scherer [2012] and Abel et al. [2016].

Our novel contribution is a *full formalization* of the algorithm and its proof of soundness, completeness, and termination, in Martin-Löf Type Theory [Martin-Löf 1975] with intensional equality, plus some well-understood extensions such as induction-recursion [Dybjer 2000]. As a mechanization language, we use Agda itself in its latest version, using the language variant without Streicher's Axiom K [1993].[1] Consequently, our proof is directly transferable to related foundations such as Homotopy Type Theory extended by induction-recursion.

The Agda formalization is quite sizeable: around 10.000 lines of code (500.000 characters). Especially the proof of the fundamental theorem of logical relations is substantial (5.000 lines). The congruence rule for the recursor for natural numbers alone requires a lemma that stretches over more than 500 lines. It is not that the proof is mathematically deep, once the right definition of the logical relation and the right formulation of the fundamental theorem are in place—it is just that a formalization requires us to get all the technicalities right. In research articles with pen and paper proofs only, the proof of the fundamental theorem is often skipped or reduced to the single sentence "proof by induction on the typing and equality derivations". Yet checking that each case of the induction goes through would require a reviewer many hours of disciplined technical reasoning. Written out, the proof would stretch over many pages.

In previous works [Abel et al. 2007; Abel and Scherer 2012; Harper and Pfenning 2005], two logical relations and two fundamental theorems are needed for the meta theory: one that entails soundness of the algorithmic equality, and one for completeness. While in pen-and-paper proofs we could get away with remarks like "proof analogous", a formalization would require us to do the proof exercise a second time; in our case, a truly intimidating task. Instead, we have been able to formulate a more abstract version of the fundamental theorem which we instantiate twice. The properties of judgemental equality which we obtain for the first instantiation are actually necessary to establish the preconditions for the second instantiation; a single, most general instantiation of the fundamental theorem is not possible to the best of our knowledge.

The abstract version of the fundamental theorem requires a logical relation parametrized on a generic notion of typed equality that is specified by 8 properties (see Section 3.1). We were able to extract these conditions by evolving the original, specific version of fundamental theorem and logical relation into the abstract one. Here, we have been harvesting the first fruit of our formalization: Once a proof is formalized, it can be safely *refactored* like a piece of software to make it more general. The mechanical proof checker ensures we are not introducing mistakes during the factorization. Our abstract formulation of the fundamental lemma is thus not only a necessity, but also an outcome of the formalization. We have been able to advance the meta-theory of dependent types by our formalization efforts.

The simpler proof technique of Harper and Pfenning [2005] which considers only the approximate shape of types by erasing the dependencies is not applicable in our case because our types might be determined by computations involving those dependencies, e.g., by recursion on natural numbers. An alternative technique to prove decidability of conversion is Normalization by Evaluation [Abel et al. 2007], however reducing terms to normal form before comparing them is often wasteful, so such a proof technique would not directly prove the correctness of a practical conversion checking algorithm. Highly related is the work of Barras [2010] which formalizes impredicative type theory in set theory which is in turn formalized in the type-theoretic proof assistent Coq [INRIA 2017]. Barras uses axiomatic inaccessible cardinals to model universes; we try here to cut out the set theory and formalize type theory directly in type theory. Altenkirch and Kaposi [2016; 2017] formalize type

[1] `agda --without-K` implements pattern matching without the K axiom.

theory in type theory using intrinsically well-typed object syntax via quotient inductive types in the meta-language. They prove normalization by evaluation in Agda. However, their formalization is less comprehensive than ours: their object type theory lacks inductive types, and their universe lacks function types.

Overview. Logical relations have been used for many purposes and with equally varied definitions, here we try to give an overview of the design principle used for ours. After specifying the syntax and rules of our type theory $\lambda^{\Pi\mathbb{U}\mathbb{N}}$, in Section 2 we prove only a minimal amount of properties by direct induction on typing or equality judgements. In particular, we prove the weakening lemma, i.e., that all the judgments still hold if we apply well-typed renamings. The remaining properties are derived from the logical relation later. We believe this makes the proof more extensible and resilient to changes in the formulation of the typing rules. In Section 2 we also define a *typed* weak head reduction relation which we prove deterministic. Using a typed variant of reduction gives us the soundness of reduction immediately, which is otherwise established by the subject reduction theorem. Subject reduction relies on the injectivity of the function type constructor which is difficult to prove for a dependent type theory with universes. In fact, it is only a consequence of the logical relation argument. Further, typed reduction is more flexible than untyped reduction and could be equipped with type-directed reduction rules needed in extensions of type theory, for instance, by singleton types or strict equality.

In Section 3 we define a Kripke logical relation, i.e., for each judgment $\Gamma \vdash J$ we define a corresponding relation $\Gamma \Vdash J$ which exhibits the inductive structure of types and their normalization properties. Many expected consequences of the judgments are actually non-trivial to derive. The logical relation rationalizes their meaning by focusing on which observations are supposed to make sense for each term. So for example $\Gamma \Vdash t : T$ not only tells us that $\Gamma \vdash t : T$ but that the same judgment holds respectively for the weak head normal forms, a and A, of t and T, and that the possible observations of a also belong to the logical relation. What we mean by observation depends on the type: if A is the universe then a has to either be neutral or a type former whose subterms also belong in the logical relation, if A is the type of natural numbers then a must be neutral or either zero or suc t for a t in the logical relation, and finally if A is the type of dependent functions, then applying a to a term in the logical relation must produce another such term. Equality judgments are similarly refined by comparing how the weak head normal forms of the terms involved react to observations. In Section 4 we present the conversion algorithm and use the consequences of the fundamental theorem to prove its termination. We prove the properties necessary to instantiate the logical relation with the conversion algorithm and use the fundamental theorem to derive its completeness. With this we can derive the decidability of the conversion judgments, which proves the conversion algorithm's correctness.

In summary, our work makes the following contribution to the programming language and type theory:

- A complete formalization of the decidability of conversion for a dependent type theory with one inductive type and one universe.
- Meta-theory based on typed weak head reduction.
- A single inductive-recursive Kripke logical relation which can be instantiated twice to first prove soundness and then completeness of the conversion algorithm. As a condition of the definition, the logical relation is indexed by a semantic type derivation, but we show proof irrelevance for these derivations.

We have restricted our investigation to the minimal core of type theory which gives us large elimination, to expose the structure of the metatheoretical development in its pure form.[2] We

[2] In contrast, the development of Abel and Scherer [2012] is slightly veiled by in addition of an irrelevance modality.

believe that our development can serve as a model for the justification and meta-theoretical investigations of extensions of type theory.

This paper is best **read in a PDF viewer**, because definitions and lemmata in blue are clickable and will open the corresponding Agda code in a browser, which is available online.[3]

2 A CORE TYPE THEORY WITH ONE UNIVERSE

In this section, we introduce $\lambda^{\Pi\mathsf{U}\mathbb{N}}$, a dependent type theory with natural numbers and recursion, dependent function types, and one universe. Using the recursor into the universe, we can define types whose shape depends on a value, for instance, the type of functions of arity $n \in \mathbb{N}$. Such recursively defined types are sometimes called *large eliminations* [Werner 1992]; their presence makes the type theory *fully dependent* in the sense that value-dependencies cannot be erased from types when constructing a model.[4]

2.1 Syntax

The grammar in Fig. 1 describes the raw syntax of $\lambda^{\Pi\mathsf{U}\mathbb{N}}$ in de Bruijn style. Expressions $t \in \mathrm{Exp}$ may or may not be in weak head normal form (Whnf), which in turn may or may not be neutral (Ne). An expression in Whnf is an expression that cannot be further reduced by the weak head reduction rules which we will later present. Expressions which are not in Whnf can all be deterministically reduced by these rules. Neutral expressions have a variable in head position that blocks further reductions.

In the formalization, Whnf and neutral are formalized as predicates over expressions. This allows us to use a simple data structure for expressions and, thanks to Agda's dependent pattern matching, it is easy to inspect them.

$\mathbb{N}$	$\ni$	x			de Bruijn indices
Exp	$\ni$	t, u, v, A, B	::=	$\bar{t} \mid t\,u \mid \mathsf{natrec}\,A\,t\,u\,v$	expressions
Whnf	$\ni$	$\bar{t}$	::=	$n \mid \mathsf{suc}\,t \mid \mathsf{zero} \mid \lambda t \mid \mathsf{U} \mid \mathbb{N} \mid \Pi\,A\,B$	weak head normal forms
Ne	$\ni$	n, m, N, M	::=	$\mathsf{i}_x \mid n\,t \mid \mathsf{natrec}\,A\,t\,u\,n$	neutral expressions
Cxt	$\ni$	Γ, Δ	::=	$\epsilon \mid \Gamma, A$	contexts
Wk	$\ni$	ρ	::=	$\mathsf{id} \mid \uparrow\rho \mid \Uparrow\rho \mid \rho \circ \rho'$	weakenings
Subst	$\ni$	σ	::=	$\rho \mid \uparrow\sigma \mid \Uparrow\sigma \mid \sigma \circ \sigma' \mid \sigma, t$	substitutions

Fig. 1. Grammar of $\lambda^{\Pi\mathsf{U}\mathbb{N}}$.

Expressions consist of the functional expressions: function application $t\,u$, abstraction λt, dependent function type $\Pi\,A\,B$; as well as the natural number expressions: zero, successor $\mathsf{suc}\,t$, natural number type $\mathbb{N}$, natural number recursion $\mathsf{natrec}\,A\,t\,u\,v$; and variables i_x and universe type U. We use $\boxed{t \equiv u}$ to denote syntactical equality of terms, which corresponds to propositional equality in the Agda formalization. For variables, we use de Bruijn [1972] indices i_x with $x \in \mathbb{N}$. The following positions bind one de Bruijn index: the sole argument of λ, the second argument of Π and the first argument of natrec. Note that the formalization does not enforce well-scopedness of expressions, instead we rely on the typing judgments to implicitly guarantee well-scopedness. In practice this has allowed for some mistakes when formalizing the typing rules, which we had to go back and correct, so intrinsically well-scoped syntax might have been a better choice.

[3] https://mr-ohman.github.io/logrel-mltt/decofconv/

[4] An example for a not fully-dependent type theory would be the Calculus of Constructions [Coquand and Huet 1988] which can be erased to F^{ω} [Geuvers 1994].

Expressions encompass values and types. For instance, $\lambda(\lambda \mathsf{i}_0)$, which would be $\lambda A.\, \lambda x.\, x$ with variable names, is the polymorphic identity function of type $\Pi\, \mathsf{U}\, (\Pi\, \mathsf{i}_0\, \mathsf{i}_1)$, which is $\Pi(A{:}\mathsf{U}).\, \Pi(x{:}A).\, A$ with names. Some expressions serve both as an object (term) and a type. For instance, $\mathbb{N}$ is the type of zero but also an inhabitant of universe U.

Contexts Γ are snoc-lists of (type) expressions to record the types of the free variables of a term or its type. We number the entries in contexts from right to left. For instance, the 3-element context (ϵ, C, B, A), associates the variables i_0 and i_1 and i_2 with types A and B and C, respectively.

Weakenings ρ, if executed on a term t as $t[\rho]$, can raise free de Bruijn indices of term t. The identity weakening id does nothing, the shifting of a weakening $\uparrow\rho$ adds one to all indices, the lifting of a weakening $\Uparrow\rho$ is for traversing under binders, and composition $\rho \circ \rho'$ (pronounce: ρ *after* ρ') lets us execute first weakening ρ' and then ρ.

Substitutions σ, executed as $t[\sigma]$, replace the free de Bruijn indices of term t by new terms. The substitution action $t[\sigma]$ is defined by recursion on the term, in Fig. 2 we give the relevant rewrite rules that follow from that. For instance, substitution (σ, t) would replace variable i_0 by t and the others according to σ. We abbreviate the singleton substitution action $t[\mathsf{id}, u]$ by $t[u]$; it just replaces i_0 by u and leaves the other variables unchanged. Weakenings are implicitly coerced to substitutions; this defines $t[\rho]$. Weakenings and substitutions obey the usual laws, for instance, $\Uparrow\sigma \circ \uparrow\mathsf{id} = \uparrow\mathsf{id} \circ \sigma$.

$$
\begin{array}{rcl}
\mathsf{U}[\sigma] & \Longrightarrow & \mathsf{U} \\
(\Pi\, F\, G)[\sigma] & \Longrightarrow & \Pi\, F[\sigma]\, G[\Uparrow\sigma] \\
\mathbb{N}[\sigma] & \Longrightarrow & \mathbb{N} \\
(\lambda t)[\sigma] & \Longrightarrow & \lambda\, t[\Uparrow\sigma] \\
(t\, u)[\sigma] & \Longrightarrow & t[\sigma]\, u[\sigma] \\
\mathsf{zero}[\sigma] & \Longrightarrow & \mathsf{zero} \\
(\mathsf{suc}\, t)[\sigma] & \Longrightarrow & \mathsf{suc}\, t[\sigma] \\
(\mathsf{natrec}\, A\, t\, u\, a)[\sigma] & \Longrightarrow & \mathsf{natrec}\, A[\Uparrow\sigma]\, t[\sigma]\, u[\sigma]\, a[\sigma]
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{i}_x\ [\mathsf{id}] & \Longrightarrow & \mathsf{i}_x \\
\mathsf{i}_x\ [\uparrow\mathsf{id}] & \Longrightarrow & \mathsf{i}_{x+1} \\
\mathsf{i}_x\ [\uparrow\sigma] & \Longrightarrow & (\mathsf{i}_x[\sigma])[\uparrow\mathsf{id}] \\
\mathsf{i}_0\ [\Uparrow\sigma] & \Longrightarrow & \mathsf{i}_0 \\
\mathsf{i}_{x+1}[\Uparrow\sigma] & \Longrightarrow & (\mathsf{i}_x[\sigma])[\uparrow\mathsf{id}] \\
\mathsf{i}_0\ [\sigma, t] & \Longrightarrow & t \\
\mathsf{i}_{x+1}[\sigma, t] & \Longrightarrow & \mathsf{i}_x[\sigma] \\
\mathsf{i}_x\ [\sigma \circ \sigma'] & \Longrightarrow & (\mathsf{i}_x[\sigma'])[\sigma]
\end{array}
$$

Fig. 2. Rewrite rules for substitutions.

For non-dependent function types, we introduce the arrow notation. We define it as a Π type with its second element weakened:

$$A \to B \triangleq \Pi\, A\, B[\uparrow\mathsf{id}]$$

Observe that $(A \to B)[\sigma] = (\Pi\, A\, B[\uparrow\mathsf{id}])[\sigma] \Longrightarrow \Pi\, A[\sigma]\, (B[\uparrow\mathsf{id}][\Uparrow\sigma]) = \Pi\, A[\sigma]\, (B[\sigma][\uparrow\mathsf{id}]) = A[\sigma] \to B[\sigma]$ as expected, according to the substitution laws.

The expression $\mathsf{natrec}\, \mathsf{U}\, A\, (\lambda\lambda(\mathbb{N} \to \mathsf{i}_0))\, n$, with names $\mathsf{natrec}\, \mathsf{U}\, A\, (\lambda m.\lambda B.\, \mathbb{N} \to B)\, n$, codes type $\mathbb{N}^n \to A$ which is short for $\mathbb{N} \to (\cdots \to (\mathbb{N} \to A) \ldots)$ with n occurrences of $\mathbb{N}$. It is an example of a large elimination of value n into universe U, producing a (small) type.

In the remainder of this paper, we reserve and exclusively use names n, m, N, M for neutral expressions, Γ, Δ for contexts, ρ for weakenings and σ for substitutions. Other names, for instance t, u, A, B, are used for expressions. Names with a bar, for instance $\bar{t}$, are used for expressions in Whnf. Note that symbols t and $\bar{t}$ are distinct, and $\bar{t}$ need not necessarily denote the Whnf of t. However, we will often use them in that way.

$\boxed{\vdash \Gamma}$ $\boxed{\Gamma \vdash A}$

$$\frac{}{\vdash \epsilon} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, A} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N}} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G}{\Gamma \vdash \Pi\, F\, G} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A}$$

$\boxed{i : A \in \Gamma}$ $\boxed{\Gamma \vdash A = B}$

$$\frac{}{\mathsf{i}_0 : A[\uparrow \mathrm{id}] \in \Gamma, A} \qquad \frac{\mathsf{i}_x : A \in \Gamma}{\mathsf{i}_{x+1} : A[\uparrow \mathrm{id}] \in \Gamma, B} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A = A} \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A} \qquad \frac{\Gamma \vdash A = B \qquad \Gamma \vdash B = C}{\Gamma \vdash A = C}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F = H \qquad \Gamma, F \vdash G = E}{\Gamma \vdash \Pi\, F\, G = \Pi\, H\, E} \qquad \frac{\Gamma \vdash A = B : \mathsf{U}}{\Gamma \vdash A = B}$$

$\boxed{\Gamma \vdash t : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : \mathsf{U}} \qquad \frac{\Gamma \vdash F : \mathsf{U} \qquad \Gamma, F \vdash G : \mathsf{U}}{\Gamma \vdash \Pi\, F\, G : \mathsf{U}} \qquad \frac{\vdash \Gamma \qquad i : A \in \Gamma}{\Gamma \vdash i : A} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G}{\Gamma \vdash \lambda t : \Pi\, F\, G}$$

$$\frac{\Gamma \vdash g : \Pi\, F\, G \qquad \Gamma \vdash a : F}{\Gamma \vdash g\, a : G[a]} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathrm{suc}\, t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathrm{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (G \to G[\uparrow \mathrm{id}, \mathrm{suc}\, \mathsf{i}_0]) \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathrm{natrec}\, G\, z\, s\, t : G[t]}$$

$\boxed{\Gamma \vdash t = u : A}$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A} \qquad \frac{\Gamma \vdash t_1 = t_2 : A \qquad \Gamma \vdash t_2 = t_3 : A}{\Gamma \vdash t_1 = t_3 : A} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t = u : B}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F = H : \mathsf{U} \qquad \Gamma, F \vdash G = E : \mathsf{U}}{\Gamma \vdash \Pi\, F\, G = \Pi\, H\, E : \mathsf{U}} \qquad \frac{\Gamma \vdash f = g : \Pi\, F\, G \qquad \Gamma \vdash a = b : F}{\Gamma \vdash f\, a = g\, b : G[a]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash f : \Pi\, F\, G \qquad \Gamma \vdash g : \Pi\, F\, G \qquad \Gamma, F \vdash f[\uparrow \mathrm{id}]\, \mathsf{i}_0 = g[\uparrow \mathrm{id}]\, \mathsf{i}_0 : G}{\Gamma \vdash f = g : \Pi\, F\, G} \qquad \frac{\Gamma \vdash t = u : \mathbb{N}}{\Gamma \vdash \mathrm{suc}\, t = \mathrm{suc}\, u : \mathbb{N}}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G \qquad \Gamma \vdash a : F}{\Gamma \vdash (\lambda t)\, a = t[a] : G[a]} \qquad \frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathrm{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathrm{id}, \mathrm{suc}\, \mathsf{i}_0])}{\Gamma \vdash \mathrm{natrec}\, F\, z\, s\, \mathrm{zero} = z : F[\mathrm{zero}]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathrm{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathrm{id}, \mathrm{suc}\, \mathsf{i}_0]) \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathrm{natrec}\, F\, z\, s\, (\mathrm{suc}\, t) = (s\, t)\, (\mathrm{natrec}\, F\, z\, s\, t) : F[\mathrm{suc}\, t]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F = F' \qquad \Gamma \vdash z = z' : F[\mathrm{zero}] \qquad \Gamma \vdash s = s' : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathrm{id}, \mathrm{suc}\, \mathsf{i}_0]) \qquad \Gamma \vdash t = t' : \mathbb{N}}{\Gamma \vdash \mathrm{natrec}\, F\, z\, s\, t = \mathrm{natrec}\, F'\, z'\, s'\, t' : F[t]}$$

Fig. 3. Inference rules of $\lambda^{\Pi \mathsf{U} \mathbb{N}}$.

$\boxed{\Gamma \vdash A \longrightarrow B}$ and $\boxed{\Gamma \vdash t \longrightarrow u : A}$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G \qquad \Gamma \vdash a : F}{\Gamma \vdash (\lambda t)\, a \longrightarrow t[a] : G[a]} \qquad \frac{\Gamma \vdash f \longrightarrow g : \Pi\, F\, G \qquad \Gamma \vdash a : F}{\Gamma \vdash f\, a \longrightarrow g\, a : G[a]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathsf{id}, \mathsf{suc}\, i_0])}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, \mathsf{zero} \longrightarrow z : F[\mathsf{zero}]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathsf{id}, \mathsf{suc}\, i_0]) \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, (\mathsf{suc}\, t) \longrightarrow (s\, t)\, (\mathsf{natrec}\, F\, z\, s\, t) : F[\mathsf{suc}\, t]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathsf{id}, \mathsf{suc}\, i_0]) \qquad \Gamma \vdash t \longrightarrow u : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, t \longrightarrow \mathsf{natrec}\, F\, z\, s\, u : F[t]}$$

$$\frac{\Gamma \vdash A \longrightarrow B : \mathsf{U}}{\Gamma \vdash A \longrightarrow B} \qquad \frac{\Gamma \vdash t \longrightarrow u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t \longrightarrow u : B}$$

$\boxed{\Gamma \vdash A \longrightarrow^* B}$ and $\boxed{\Gamma \vdash t \longrightarrow^* u : A}$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \longrightarrow^* A} \qquad \frac{\Gamma \vdash A \longrightarrow B \qquad \Gamma \vdash B \longrightarrow^* C}{\Gamma \vdash A \longrightarrow^* C} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash t \longrightarrow^* t : A} \qquad \frac{\Gamma \vdash t \longrightarrow u : A \qquad \Gamma \vdash u \longrightarrow^* a : A}{\Gamma \vdash t \longrightarrow^* a : A}$$

Fig. 4. Weak head reduction rules.

2.2 Rules and Semantics

In Fig. 3 we define the judgements $\boxed{\vdash \Gamma}$ for well-formed contexts, $\boxed{\Gamma \vdash A}$ for well-formed types, $\boxed{\Gamma \vdash A = B}$ for conversion of types, $\boxed{\Gamma \vdash t : A}$ for type membership and $\boxed{\Gamma \vdash t = u : A}$ for conversion of terms. These are all defined simultaneously: If we look at the following rules, we can see how the judgements depend on each other:

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A = B : \mathsf{U}}{\Gamma \vdash A = B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t = u : B}$$

The first two rules let us lift a term or a term equality to the type level if it is of type U. This shows us that the judgements for types, $\Gamma \vdash A$ and $\Gamma \vdash A = B$, depend on the judgements for terms $\Gamma \vdash t : A$ and $\Gamma \vdash t = u : A$, respectively. The last two rules are the conversion rules, which let us take a term or a term equality to another, equivalent type. Here we see that both $\Gamma \vdash t : A$ and $\Gamma \vdash t = u : A$ depend on $\Gamma \vdash A = B$. Additionally, with the reflexivity rules, we can see that the equality judgements refer to the typing judgements, thus, all the judgements depend on each other and need to be defined simultaneously.

In Fig. 4 we list the reduction rules of $\lambda^{\Pi \mathsf{U} \mathbb{N}}$. Judgement $\boxed{\Gamma \vdash t \longrightarrow u : A}$ performs a single weak head reduction step from t to u, and $\boxed{\Gamma \vdash t \longrightarrow^* u : A}$ is its reflexive-transitive closure. Our grammar for Whnf captures exactly the well-typed terms that cannot be reduced further with these judgements. Any well-typed term not in Whnf will in finitely many steps reduce to a term in Whnf, yet this fact requires proof by logical relation and will appear quite late in our meta-theoretic development, see the Weak Head Normalization Theorem (3.28). Weak head normalization allows us to find the canonical head constructor of an expression if there is one.

Note that the reduction rules are typed, which is one of the main technical innovations of Abel, Coquand, and Manna [2016]. We immediately get that reduction is included in conversion. In contrast, with untyped reduction, we would need a type preservation (aka subject reduction) theorem which requires function type injectivity (aka Π injectivity) which in turn needs proof by a logical relation [Abel et al. 2007; Abel and Scherer 2012; Goguen 1994].

Below, we prove some key properties of the rules.

Lemma 2.1 (Well-formed context). *If* $\Gamma \vdash J$ *then* $\vdash \Gamma$ *for any typing or conversion judgement* J.

Proof. By induction on the judgement. □

Lemma 2.2 (Reduction subsumed by equality).

(1) *If either* $\Gamma \vdash A \longrightarrow B$ *or* $\Gamma \vdash A \longrightarrow^* B$ *then* $\Gamma \vdash A = B$.
(2) *If either* $\Gamma \vdash t \longrightarrow u : A$ *or* $\Gamma \vdash t \longrightarrow^* u : A$ *then* $\Gamma \vdash t = u : A$.

Proof. By induction on the judgement. □

With typing, we can also show that the first element of a reduction is a well-formed type or term.

Lemma 2.3 (Subject typing).

(1) *If* $\Gamma \vdash A \longrightarrow B$ *then* $\Gamma \vdash A$.
(2) *If* $\Gamma \vdash t \longrightarrow u : A$ *then* $\Gamma \vdash t : A$.

Proof. By induction on the reduction and by the well-formedness of the context (2.1). □

The analogue of this lemma, to derive $\Gamma \vdash B$ from $\Gamma \vdash A \longrightarrow B$ and $\Gamma \vdash u : A$ from $\Gamma \vdash t \longrightarrow u : A$, will be proven as a consequence of the fundamental theorem (see Thm. 3.26).

Lemma 2.4 (Whnfs do not reduce). *We cannot step from a weak head normal form.*

(1) $\Gamma \vdash \overline{A} \longrightarrow A'$ *is impossible.*
(2) $\Gamma \vdash \overline{t} \longrightarrow t' : B$ *is impossible.*

Furthermore, a reduction sequence starting with a Whnf *goes nowhere:*

(3) *If* $\Gamma \vdash \overline{A} \longrightarrow^* A'$ *then* $\overline{A} \equiv A'$.
(4) *If* $\Gamma \vdash \overline{t} \longrightarrow^* t' : C$ *then* $\overline{t} \equiv t'$.

Proof. By induction on the reduction. □

Reduction is deterministic; each expression has at most one reduct.

Lemma 2.5 (Reduction is deterministic).

(1) *If* $\Gamma \vdash A \longrightarrow B$ *and* $\Gamma \vdash A \longrightarrow B'$ *then* $B \equiv B'$.
(2) *If* $\Gamma \vdash t \longrightarrow u : A$ *and* $\Gamma \vdash t \longrightarrow u' : A$ *then* $u \equiv u'$.
(3) *If* $\Gamma \vdash A \longrightarrow^* \overline{A}$ *and* $\Gamma \vdash A \longrightarrow^* \overline{A'}$ *then* $\overline{A} \equiv \overline{A'}$.
(4) *If* $\Gamma \vdash t \longrightarrow^* \overline{t} : A$ *and* $\Gamma \vdash t \longrightarrow^* \overline{t'} : A$ *then* $\overline{t} \equiv \overline{t'}$.

Proof. By induction on the reduction, using the fact that Whnfs do not reduce (2.4). □

We classify the weakenings ρ from Γ to Δ by judgement $\boxed{\rho : \Delta \leq \Gamma}$, inductively given by the rules to follow. If we apply a well-formed weakening $\rho : \Delta \leq \Gamma$ to a term t in Γ, we obtain a term $t[\rho]$ in Δ.

$$\frac{}{\mathsf{id} : \Gamma \leq \Gamma} \qquad \frac{\rho : \Delta \leq \Gamma}{\uparrow \rho : (\Delta, A) \leq \Gamma} \qquad \frac{\rho : \Delta \leq \Gamma}{\Uparrow \rho : (\Delta, A[\rho]) \leq (\Gamma, A)}$$

The rule for weakening composition is missing, because it is admissible.

Lemma 2.6 (Weakening composition). *If* $\rho : \Delta' \leq \Delta$ *and* $\rho' : \Delta \leq \Gamma$ *then* $\rho \circ \rho' : \Delta' \leq \Gamma$.

Proof. By induction on the structure of well-formed weakenings. □

In the remainder of this article, we shall implicitly assume well-formedness of both contexts, $\vdash \Delta$ and $\vdash \Gamma$, whenever we mention $\rho : \Delta \leq \Gamma$.

To prove the Weakening Lemma (2.7) we strengthened our typing rules slightly, leading to the hypotheses marked in grey. For instance, in the case of $\Gamma \vdash \lambda t : \Pi\, F\, G$, it is not enough to only have $\Gamma, F \vdash t : G$ since we also need $\Gamma \vdash F$. While we can extract $\Gamma \vdash F$ from $\Gamma, F \vdash t : G$ via the context $\vdash \Gamma, F$, it does not immediately follow that the extracted derivation of $\Gamma \vdash F$ is a smaller than the original derivation of $\Gamma \vdash \lambda t : \Pi\, F\, G$, which means that the use of the induction hypothesis would not be justified a priori. Hence, we have strengthened the premises of our rules by an additional hypothesis $\Gamma \vdash F$ whenever the context extension Γ, F appears in a hypothesis. A similar technique has been applied by Harper and Pfenning [2005].

Alternatively, we could use sized types in the metalanguage [Abel and Pientka 2016; Hughes et al. 1996; Sacchini 2013]. This would involve making our judgments sized, so that when we extract $\Gamma \vdash F$ from $\Gamma, F \vdash t : G$ via context $\vdash \Gamma, F$, the sizes would witness that $\Gamma \vdash F$ is smaller than $\vdash \Gamma, F$ which is in turn smaller than $\Gamma, F \vdash t : G$. Thus, we could justify the use of the induction hypothesis on $\Gamma \vdash F$.

Lemma 2.7 (Weakening). *Let* $\rho : \Delta \leq \Gamma$. *If* $\Gamma \vdash J$ *then* $\Delta \vdash J[\rho]$ *for any typing, conversion, or reduction judgement* J.

Proof. By induction on the judgement. □

Finally, we define well-formed substitutions and their equality as $\boxed{\Delta \vdash \sigma : \Gamma}$ and $\boxed{\Delta \vdash \sigma = \sigma' : \Gamma}$, inductively by the rules to follow.

$$\frac{}{\Delta \vdash \sigma : \epsilon} \qquad \frac{\Delta \vdash \sigma \circ \uparrow\mathsf{id} : \Gamma \qquad \Delta \vdash i_0[\sigma] : A[\sigma \circ \uparrow\mathsf{id}]}{\Delta \vdash \sigma : \Gamma, A}$$

$$\frac{}{\Delta \vdash \sigma = \sigma' : \epsilon} \qquad \frac{\Delta \vdash \sigma \circ \uparrow\mathsf{id} = \sigma' \circ \uparrow\mathsf{id} : \Gamma \qquad \Delta \vdash i_0[\sigma] = i_0[\sigma'] : A[\sigma \circ \uparrow\mathsf{id}]}{\Delta \vdash \sigma = \sigma' : \Gamma, A}$$

We may consider a non-empty substitution $\Delta \vdash \sigma : \Gamma, A$ as a pair $(\sigma \circ \uparrow\mathsf{id},\ i_0[\sigma])$ of a substitution $\sigma \circ \uparrow\mathsf{id}$ and a term $i_0[\sigma]$, where we call the second component the *head* of σ and the first component the *tail* of σ.

Similarly to well-formed weakenings, we shall henceforth assume that the contexts Γ and Δ are well-formed when we mention $\Delta \vdash \sigma : \Gamma$ or $\Delta \vdash \sigma = \sigma' : \Gamma$. Unlike well-formed weakenings, we do not need to immediately prove that well-formed substitutions can be applied to well-formed expressions to create new well-formed expressions. Instead, this will follow from the fundamental theorem (see Thm. 3.31).

3 KRIPKE LOGICAL RELATIONS

To prove decidability of $\lambda^{\Pi\mathbb{U}\mathbb{N}}$'s judgemental equality, we will show that it is equivalent to a more structured equality relation, called *algorithmic equality* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$. For one, algorithmic equality will have no rule for transitivity, because the transitivity rule is very non-deterministic: If searching for a derivation of $\Gamma \vdash t = v : A$ we have to find a u such that $\Gamma \vdash t = u : A$ and $\Gamma \vdash u = v : A$, it is not clear how to guarantee progress. Instead, transitivity for algorithmic

equality will be admissible. Secondly, we will prove that to derive $\Gamma \vdash t = u : A$, it is sufficient to derive equality $\Gamma \vdash \overline{t} = \overline{u} : A$ of the weak head normal forms of t and u. This way, we can exploit the structure of objects during equality check; for instance, *injectivity of constructors*: $\Gamma \vdash \mathsf{suc}\, t = \mathsf{suc}\, t' : \mathbb{N}$ holds iff $\Gamma \vdash t = t' : \mathbb{N}$.

For $\lambda^{\Pi\mathbb{U}\mathbb{N}}$ and related type theories, the essential properties like weak head normalization and injectivity of constructors are not provable directly by induction on the typing and equality judgements. Instead a *logical relation* is needed which represents the structure of objects explicitely, such as

- *canonicity*: a closed term of type $\mathbb{N}$ reduces either to zero or $\mathsf{suc}\, t$, or
- *function type injectivity*: if two function types are equal, then their domains and codomains are equal.

In this section, we will construct a logical relation $\Gamma \Vdash_\ell t = u : A$, pronounced "$t$ and u are reducibly equal at type A (of level ℓ in context Γ)." This relation will be *Kripke* in the sense that it is closed under weakening. It will expose the inductive structure of objects such that we can prove desired properties like normalization, canonicity, and injectivity. Analogously, we will define reducibility predicates and relations for the other main judgements of $\lambda^{\Pi\mathbb{U}\mathbb{N}}$, namely $\Gamma \vdash A$ and $\Gamma \vdash A = B$ and $\Gamma \vdash t : A$.

Reducible equality $\Gamma \Vdash_\ell t = u : A$ will be a subrelation of judgemental equality $\Gamma \vdash t = u : A$, but we will later need a similar relation that is a subrelation of algorithmic equality. Thus, we define it as a subrelation of a *generic equality* relation $\Gamma \vdash t \cong u : A$ which can be instantiated for both purposes. To prove that judgemental equality is in turn a subrelation of reducible equality, i.e., reducible equality is complete, we have to introduce a third relation, *valid equality* $\Gamma \Vdash^{\mathsf{v}}_\ell t = u : A$. Fig. 5 summarizes these relations and their connections.

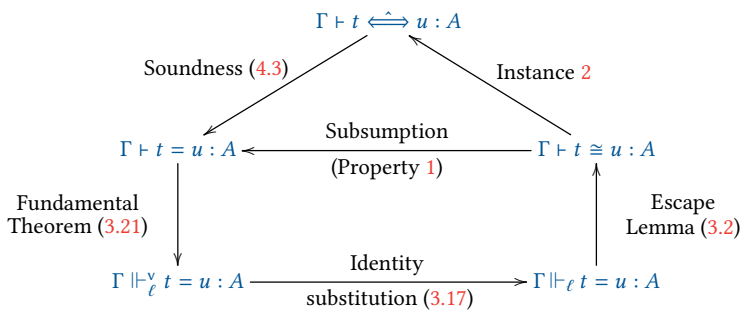

Fig. 5. Proof outline.

3.1 Generic Equality

To prove decidability of conversion, we need to introduce two logical relations, one with the conversion judgements for equality and another with algorithmic equality (see Section 4.1). The first logical relation will be used to derive proofs which are necessary to prove the properties of algorithmic equality, one being decidability, while the second logical relation will be used to prove completeness of algorithmic equality. The proofs about these logical relations are quite large, and with the two relations being quite similar, we would like to be able to not duplicate our work. We

therefore introduce a notion of generic equality which we use to parameterize the definition of our logical relation, such that it can be instantiated to get the two logical relations we need.

We here give a specification for a set of relations being a *generic equality*. A generic equality consists of three relations that satisfy the properties we list below: $\boxed{\Gamma \vdash A \cong B}$ for equality of types, $\boxed{\Gamma \vdash t \cong u : A}$ for equality of terms of a type and $\boxed{\Gamma \vdash t \sim u : A}$ for equality between neutral terms. In the formalization, this is implemented as a record type with one field for each of the three relations and one field for each property.

Property 1 (Subsumption).

(1) *If* $\Gamma \vdash t \sim u : A$ *then* $\Gamma \vdash t \cong u : A$. *(Neutral equality is included in generic equality.)*
(2) *If* $\Gamma \vdash A \cong B : \mathsf{U}$ *then* $\Gamma \vdash A \cong B$. *(Small types are included in large types.)*

Further, generic equality is a subrelation of judgmental equality.

(3) *If* $\Gamma \vdash A \cong B$ *then* $\Gamma \vdash A = B$.
(4) *If* $\Gamma \vdash t \cong u : A$ *then* $\Gamma \vdash t = u : A$.

Property 2 (Partial equivalence relation). *The three generic equality relations* $\Gamma \vdash \_ \cong \_$ *and* $\Gamma \vdash \_ \cong \_ : A$ *and* $\Gamma \vdash \_ \sim \_ : A$ *are symmetric and transitive.*

Property 3 (Conversion). *If* $\Gamma \vdash t \cong u : A$ *and* $\Gamma \vdash A = B$ *then* $\Gamma \vdash t \cong u : B$. *(Same for* $\sim$*.)*

Property 4 (Weakening). *Generic equality is closed under weakning, i.e., if* $\rho : \Delta \leq \Gamma$ *and* $\Gamma \vdash J$ *then* $\Delta \vdash J[\rho]$*, where* J *ranges over the three equality forms.*

Property 5 (Weak head expansion). $\Gamma \vdash \_ \cong \_$ *and* $\Gamma \vdash \_ \cong \_ : A$ *are closed under weak head expansion.*

(1) *If* $\Gamma \vdash A \longrightarrow^* \overline{A}$ *and* $\Gamma \vdash B \longrightarrow^* \overline{B}$ *and* $\Gamma \vdash \overline{A} \cong \overline{B}$ *then* $\Gamma \vdash A \cong B$.
(2) *If* $\Gamma \vdash A \longrightarrow^* \overline{B}$ *and* $\Gamma \vdash a \longrightarrow^* \overline{a} : B$ *and* $\Gamma \vdash b \longrightarrow^* \overline{b} : B$ *and* $\Gamma \vdash \overline{a} \cong \overline{b} : B$ *then* $\Gamma \vdash a \cong b : A$.

Property 6 (Type constructor congruence). *If* $\vdash \Gamma$ *then:*

(1) $\Gamma \vdash \mathsf{U} \cong \mathsf{U}$.
(2) $\Gamma \vdash \mathbb{N} \cong \mathbb{N}$ *and* $\Gamma \vdash \mathbb{N} \cong \mathbb{N} : \mathsf{U}$.
(3) *If* $\Gamma \vdash F \cong H$ *and* $\Gamma, F \vdash G \cong E$ *then* $\Gamma \vdash \Pi\, F\, G \cong \Pi\, H\, E$. *(And analogously for* $\Gamma \vdash \_ \cong \_ : \mathsf{U}$*.)*

Property 7 (Value constructor congruence and η).

(1) *If* $\vdash \Gamma$ *then* $\Gamma \vdash \mathsf{zero} \cong \mathsf{zero} : \mathbb{N}$.
(2) *If* $\Gamma \vdash t \cong u : \mathbb{N}$ *then* $\Gamma \vdash \mathsf{suc}\, t \cong \mathsf{suc}\, u : \mathbb{N}$.
(3) *If* $\Gamma \vdash F$ *and* $\Gamma \vdash f : \Pi\, F\, G$ *and* $\Gamma \vdash g : \Pi\, F\, G$ *and* $\Gamma, F \vdash f[\uparrow \mathsf{id}]\, \mathsf{i}_0 \cong g[\uparrow \mathsf{id}]\, \mathsf{i}_0 : G$ *then* $\Gamma \vdash f \cong g : \Pi\, F\, G$.

Property 8 (Congruence for neutrals).

(1) *If* $\Gamma \vdash i : A$ *then* $\Gamma \vdash i \sim i : A$.
(2) *If* $\Gamma \vdash f \sim g : \Pi\, F\, G$ *and* $\Gamma \vdash a \cong b : F$ *then* $\Gamma \vdash f\, a \sim g\, b : G[a]$.
(3) *If* $\Gamma, \mathbb{N} \vdash F \cong F'$ *and* $\Gamma \vdash z \cong z' : F[\mathsf{zero}]$ *and* $\Gamma \vdash s \cong s' : \Pi\, \mathbb{N}\, (F \longrightarrow F[\uparrow \mathsf{id}, \mathsf{suc}\, \mathsf{i}_0])$ *and* $\Gamma \vdash n \sim n' : \mathbb{N}$ *then* $\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, n \sim \mathsf{natrec}\, F'\, z'\, s'\, n' : F[n]$.

Now that we have defined all the necessary properties, we will introduce our first instance of generic equality, which is simply judgmental equality.

Instance 1 (Judgmental equality). *The following instantiation of generic equality satisfies all the required properties:*

$$\Gamma \vdash A \cong B \text{ instantiated to } \Gamma \vdash A = B$$
$$\Gamma \vdash t \cong u : A \text{ instantiated to } \Gamma \vdash t = u : A$$
$$\Gamma \vdash t \sim u : A \text{ instantiated to } \Gamma \vdash t = u : A$$

Proof. Subsumption (Prop. 1) is obvious. The weakening property (Prop. 4) is satisfied by the weakening lemma (2.7) and weak head expansion (Prop. 5) is proven by the fact that reduction is subsumed by equality (2.2). The remaining properties are fulfilled by the inference rules of judgmental equality. □

Note that for definitional equality we do not need the distinction between $\Gamma \vdash t \cong u : A$ and $\Gamma \vdash t \sim u : A$. However, for the algorithmic equality it is required, therefore it is part of the specification.

3.2 A Logical Relation for Reducibility

In this section, we define *reducibility judgements* $\Gamma \Vdash_\ell J$ for types, type equality, terms, and term equality. These judgements take the form of logical predicates and relations [Friedman 1975]. They entail well-formedness, i.e., if $\Gamma \Vdash_\ell J$ then $\Gamma \vdash J$, see the Escape[5] Lemma (3.2). The opposite direction, $\Gamma \vdash J$ implies $\Gamma \Vdash_\ell J$ will be a consequence of the Fundamental Theorem of Logical Relations (3.21). A logical relation on well-typed open terms is necessarily *Kripke*, i.e., closed under weakening: if $\rho : \Delta \leq \Gamma$ and $\Gamma \Vdash_\ell J$ then $\Delta \Vdash_\ell J$. Otherwise, the Fundamental Theorem would fail for binders (λ and Π).

The index ℓ denotes the type *level* and ranges over 0, 1. We shall define the reducibility judgements by induction on ℓ, i.e., first for small types ($\ell = 0$) and then for large types ($\ell = 1$), using the judgements for small types. Spelled out, the reducibility judgements are:

$\Gamma \Vdash_\ell A$	A is a reducible type of level ℓ in context Γ
$\Gamma \Vdash_\ell A = B$	A and B are reducibly equal types of level ℓ in context Γ
$\Gamma \Vdash_\ell t : A$	t is a reducible term of level-ℓ type A in context Γ
$\Gamma \Vdash_\ell t = u : A$	t and u are reducibly equal terms of level-ℓ type A in context Γ

These judgements will imply that all involved objects are *reducible* [Girard 1972] (in the sense of weak normalization), in particular, all involved objects have a weak head normal form. Further, the judgements do not distinguish between objects that have the same weak head normal form. This is achieved by defining the judgements on weak head normal forms and closing them under weak head expansion.

The qualifier *"logical"* means that objects are characterized by their behavior, e.g., non-neutral objects of type $\mathbb{N}$ should reduce to zero or $\mathsf{suc}\, t$ for a reducible t, and functions should yield a reducible result if applied to a reducible argument. As a first approximation, let us define $\Gamma \Vdash_\ell t : \Pi\, F\, G$ to hold if $\Gamma \vdash t \longrightarrow^* \overline{t} : \Pi\, F\, G$ and for all $\Gamma \Vdash_\ell a : F$ we have $\Gamma \Vdash_\ell t\, a : G[a]$. Note that formula $\Gamma \Vdash_\ell a : F$ occurs *negatively* in this definition, which has dire consequences: First, we will not be able to inductively prove weakening for this definition, thus, we have to build it into the definition. Hence, we require instead that for all $\rho : \Delta \leq \Gamma$ and $\Delta \Vdash_\ell a : F[\rho]$ we have $\Delta \Vdash_\ell t[\rho]\, a : G[\rho, a]$.

Secondly, the negative occurrence prevents us to define $\Gamma \Vdash_\ell t : A$ as inductive predicate. Instead, we have to define it by recursion on the type A, but not simply on the size of the type expression A, since this does not get smaller in the recursive calls, e.g., $\Delta \Vdash_\ell t[\rho]\, a : G[\rho, a]$. We define $\Gamma \Vdash_\ell t : A$ by recursion on the *derivation* $\mathcal{T}$ of $\Gamma \Vdash_\ell A$, written $\mathcal{T} :: \Gamma \Vdash_\ell A$, which in turn is an inductive predicate.

[5] The terminology *escaping the logical relation* was coined by Schürmann and Sarnat [2008].

Thus, the term reducibility judgement will depend on $\mathcal{T}$ and we write $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$. In turn, the rule for $\Gamma \Vdash_\ell \Pi\, F\, G$ will have to refer to reducible terms of type F, but we may assume $\Delta \Vdash_\ell a : F/_{\mathcal{U}}$ to be already defined since $\mathcal{U} :: \Gamma \Vdash_\ell F$ is a subderivation mentioned in the premise of the rule. This definition scheme of interleaving induction and recursion is called *induction-recursion* [Dybjer 2000]. While the dependence on derivation is necessary for the well-foundedness of the definition process, the exact shape of the derivation should be irrelevant for the reducibility judgements. We prove this *a posteriori* in Lemma 3.5.

We now to proceed to give the rules for the inductive predicate $\boxed{\Gamma \Vdash_\ell A}$, each rule followed by the clauses for the judgements that are simultaneously defined by recursion on its derivation $\mathcal{T}$: $\boxed{\Gamma \Vdash_\ell A = B/_{\mathcal{T}}}$ and $\boxed{\Gamma \Vdash_\ell t : A/_{\mathcal{T}}}$ and $\boxed{\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}}$. In the following, we often refer to the package of these four judgements as *the logical relation*. The logical relation is implemented under an external well-founded induction on ℓ, so that definitions for large types can make use of the relations for small ones. Most rules are identical for both levels, as most of the type formers belong to both small and large types—except for the universe U which is necessarily a large type.

Universe. U shall be a large reducible type.

$$\frac{\vdash \Gamma}{\Gamma \Vdash_\ell \mathsf{U}}\ \ell' < \ell$$

Since we have no reduction on the level of large types, and hence, nothing reduces to U, this rule is trivially closed under weak head expansion. The side condition $\ell' < \ell$ could be written as $\ell = 1$. But since we doing a well-founded induction on ℓ, our formulation is convenient, as it directly gives us the induction hypothesis for ℓ'.

For derivations $\mathcal{T}$ built with that rule we define:

- $\Gamma \Vdash_\ell \mathsf{U} = B/_{\mathcal{T}}$ iff $B \equiv \mathsf{U}$. *This means that only* U *is reducibly equal to itself.*
- $\Gamma \Vdash_\ell t : \mathsf{U}/_{\mathcal{T}}$ *(*t *is a reducible member of* U*)* iff the following hold:
  (1) There exists some $\bar{t}$ such that $\Gamma \vdash t :\longrightarrow^* \bar{t} : \mathsf{U}$ and $\Gamma \vdash \bar{t} \cong \bar{t} : \mathsf{U}$, *meaning* t *has a reflexive whnf.*
  (2) $\Gamma \Vdash_{\ell'} t$, *which means that* t *is a reducible small type (already defined by induction hypothesis).*
- $\Gamma \Vdash_\ell t = u : \mathsf{U}/_{\mathcal{T}}$ *(*t *and* u *are reducibly equal members of* U*)* iff:
  (1) There are $\bar{t}$ and $\bar{u}$ such that $\Gamma \vdash t :\longrightarrow^* \bar{t} : \mathsf{U}$ and $\Gamma \vdash u :\longrightarrow^* \bar{u} : \mathsf{U}$ and $\Gamma \vdash \bar{t} \cong \bar{u} : \mathsf{U}$.
  This means that t *and* u *have whnfs related by the generic equality.*
  (2) $\mathcal{U} :: \Gamma \Vdash_{\ell'} t$ and $\Gamma \Vdash_{\ell'} u$ and $\Gamma \Vdash_{\ell'} t = u/_{\mathcal{U}}$, *meaning that* t *and* u *are reducibly equal small types.*

Neutrals. Any type that has a neutral whnf N shall be reducible. Since generic equality is not reflexive in general, we also require N to be reflexive.

$$\frac{\Gamma \vdash A :\longrightarrow^* N \qquad \Gamma \vdash N \sim N : \mathsf{U}}{\Gamma \Vdash_\ell A}$$

Here, $\boxed{\Gamma \vdash A :\longrightarrow^* N}$ is a short-hand for the conjunction of $\Gamma \vdash A$ and $\Gamma \vdash N$ and $\Gamma \vdash A \longrightarrow^* N$. For any relation $\circledR$, we will from now on use $\Gamma \vdash A :\circledR: B$ to signify the conjunction of $\Gamma \vdash A$ and $\Gamma \vdash B$ and $\Gamma \vdash A \circledR B$, and use $\Gamma \vdash t :\circledR: u : A$ to signify the conjunction of $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$ and $\Gamma \vdash t \circledR u : A$. We introduce the extra well-formedness premises because we cannot yet prove $\Gamma \vdash A$ or $\Gamma \vdash B$ from most of our relations, for instance, neither from $\Gamma \vdash A = B$ nor from $\Gamma \vdash A \longrightarrow^* B$. The lacking property is known as *syntactic validity* [Harper and Pfenning 2005] or *presupposition* [Goguen 2000]. It is non-trivial to derive because of the asymmetry present in some

of the inference rules, like the congruence rule for application or Π-types. We will obtain syntactic validity (Thm. 3.26) later as a consequence of the fundamental theorem.

Given a derivation $\mathcal{T} :: \Gamma \Vdash_\ell A$ built by the rule for neutral types we define:

- $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ iff there is a neutral M such that $\Gamma \vdash B :\longrightarrow^*: M$ and $\Gamma \vdash N \sim M : \mathsf{U}$.
  Neutral types are reducibly equal to types that have an equal whnf up to generic neutral equality.
- $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ iff there is a neutral n such that $\Gamma \vdash t :\longrightarrow^*: n : N$ and $\Gamma \vdash n \sim n : N$.
  Neutral types are inhabited by terms that have a neutral whnf.
- $\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}$ iff $\Gamma \vdash t :\longrightarrow^*: n : N$ and $\Gamma \vdash u :\longrightarrow^*: m : N$ and $\Gamma \vdash n \sim m : N$ for some neutrals n, m.
  At neutral type, objects are reducibly equal if they have the same whnf up to generic neutral equality.

Natural Numbers. $\mathbb{N}$ and its well-formed weak head expansions are reducible types.

$$\frac{\Gamma \vdash A :\longrightarrow^*: \mathbb{N}}{\Gamma \Vdash_\ell A}$$

For a derivation $\mathcal{T}$ built by that rule we define:

- $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ iff $\Gamma \vdash B \longrightarrow^* \mathbb{N}$.
- $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ iff $\boxed{\Gamma \Vdash \mathbb{N}\ t}$ which is defined to hold iff there exists a whnf $\bar{t}$ such that:
  (1) $\Gamma \vdash t :\longrightarrow^*: \bar{t} : \mathbb{N}$
  (2) $\Gamma \vdash \bar{t} \cong \bar{t} : \mathbb{N}$
  (3) $\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \bar{t}$, which is inductively defined by the rules

$$\frac{}{\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \mathsf{zero}} \qquad \frac{\Gamma \Vdash \mathbb{N}\ t}{\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \mathsf{suc}\ t} \qquad \frac{\Gamma \vdash n : \mathbb{N} \qquad \Gamma \vdash n \sim n : \mathbb{N}}{\Gamma \Vdash \mathbb{N}_\mathsf{w}\ n}$$

  With that inductive definition, we model the different properties of the possible constructions of natural numbers. For the $\mathsf{suc}\ t$ case, we require that t is a reducible natural number, so that we can properly use natural recursion on the number. For the neutral case, we require that n is well-formed and neutrally reflexive. Finally, for the zero case, we have no additional requirements.
- $\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}$ iff $\boxed{\Gamma \Vdash \mathbb{N}\ t = u}$ which is defined to hold iff there exist whnfs $\bar{t}$ and $\bar{u}$ such that:
  (1) $\Gamma \vdash t :\longrightarrow^*: \bar{t} : \mathbb{N}$
  (2) $\Gamma \vdash u :\longrightarrow^*: \bar{u} : \mathbb{N}$
  (3) $\Gamma \vdash \bar{t} \cong \bar{u} : \mathbb{N}$
  (4) $\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \bar{t} = \bar{u}$, which is analogously to $\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \bar{t}$ defined inductively by the rules

$$\frac{}{\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \mathsf{zero} = \mathsf{zero}} \qquad \frac{\Gamma \Vdash \mathbb{N}\ t = u}{\Gamma \Vdash \mathbb{N}_\mathsf{w}\ \mathsf{suc}\ t = \mathsf{suc}\ u} \qquad \frac{\Gamma \vdash n \sim m : \mathbb{N}}{\Gamma \Vdash \mathbb{N}_\mathsf{w}\ n = m}$$

Function Spaces. A type with whnf $\Pi\, F\, G$ is reducible if both F and G are reducible in a way detailed in the following rule:

$$\frac{\begin{array}{l}\Gamma \vdash A :\longrightarrow^*: \Pi\, F\, G \qquad \Gamma \vdash F \qquad \Gamma, F \vdash G \\ \mathcal{U} :: (\forall \rho : \Delta \leq \Gamma.\ \Delta \Vdash_\ell F[\rho]) \qquad \mathcal{V} :: (\forall \rho : \Delta \leq \Gamma.\ \Delta \Vdash_\ell a : F[\rho]/_{\mathcal{U}(\rho)} \implies \Delta \Vdash_\ell G[\rho, a]) \\ \forall \rho : \Delta \leq \Gamma,\ u :: \Delta \Vdash_\ell a : F[\rho]/_{\mathcal{U}(\rho)}. \\ \quad \Delta \Vdash_\ell b : F[\rho]/_{\mathcal{U}(\rho)} \implies \Delta \Vdash_\ell a = b : F[\rho]/_{\mathcal{U}(\rho)} \implies \Delta \Vdash_\ell G[\rho, a] = G[\rho, b]/_{\mathcal{V}(\rho, u)}\end{array}}{\Gamma \Vdash_\ell A}$$

The notation "::" denotes the assignment of a type to a variable in the meta-theory. It is the same as Agda's elementhood relation ":" and can for instance be used in $\forall u :: \mathcal{U}.\ \mathcal{V}(u)$ which means

"for all u of type $\mathcal{U}$, there is $\mathcal{V}(u)$." For weakening, we abbreviate $(\forall \rho :: \mathrm{Wk}.\ \ \rho : \Gamma \le \Delta \implies P)$ to $(\forall \rho : \Gamma \le \Delta.\ \ P)$.

For A to be a reducible Π-type, we first require A to reduce to $\Pi\, F\, G$, where F and G are well-formed. Secondly, F has to be reducible under any weakening ρ. Thirdly, for any reducible term a in $F[\rho]$, the type $G[\rho, a]$ has to be reducible. Finally, if a and b are reducibly equal of type $F[\rho]$, the types $G[\rho, a]$ and $G[\rho, b]$ are reducibly equal as well. The requirement of reducibility under weakening is needed for proving reducibility under binder cases like λ and Π, and the last two conditions model that G is a F-indexed family of types.

Given a derivation $\mathcal{T}$ built by the function space rule we define:

- $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ iff there are F' and G' such that:

(1) $\Gamma \vdash B \longrightarrow^* \Pi\, F'\, G'$
(2) $\Gamma \vdash \Pi\, F\, G \cong \Pi\, F'\, G'$
(3) $\forall \rho : \Delta \le \Gamma.\ \ \Delta \Vdash_\ell F[\rho] = F'[\rho]/_{\mathcal{U}(\rho)}$
(4) $\forall \rho : \Delta \le \Gamma,\ u :: \Delta \Vdash_\ell a : F[\rho]/_{\mathcal{U}(\rho)}.\ \ \Delta \Vdash_\ell G[\rho, a] = G'[\rho, a]/_{\mathcal{V}(\rho,u)}$

For B to be reducibly equal to A, the type B has to reduce to $\Pi\, F'\, G'$ which is equal in generic equality to the $\Pi\, F\, G$, the type A reduces to. We also require that under weakening ρ, the types $F[\rho]$ and $F'[\rho]$ have to be reducibly equal and for a of type $F[\rho]$, $G[\rho, a]$ and $G'[\rho, a]$ are reducibly equal.

- $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ iff there is a $\overline{t}$ such that:

(1) $\Gamma \vdash t :\longrightarrow^*: \overline{t} : \Pi\, F\, G$
(2) $\Gamma \vdash \overline{t} \cong \overline{t} : \Pi\, F\, G$
(3) $\forall \rho : \Delta \le \Gamma,\ u :: \Delta \Vdash_\ell a : F[\rho]/_{\mathcal{U}(\rho)}.\ \ \Delta \Vdash_\ell \overline{t}[\rho]\, a : G[\rho, a]/_{\mathcal{V}(\rho,u)}$
(4) $\forall \rho : \Delta \le \Gamma,\ u :: \Delta \Vdash_\ell a : F[\rho]/_{\mathcal{U}(\rho)}.\ \ \Delta \Vdash_\ell b : F[\rho]/_{\mathcal{U}(\rho)} \implies \Delta \Vdash_\ell a = b : F[\rho]/_{\mathcal{U}(\rho)} \implies \Delta \Vdash_\ell \overline{t}[\rho]\, a = \overline{t}[\rho]\, b : G[\rho, a]/_{\mathcal{V}(\rho,u)}$

The requirement 3 effectively says that under weakening ρ, given a term a of type $F[\rho]$, we can apply a to $\overline{t}[\rho]$ with type $G[\rho, a]$. Requirement 4 basically says the same thing, but for equality. Also note that the reducibility of b is not necessary to complete the definition, but will be helpful in some of the proofs.

- $\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}$ iff there is a $\overline{t}$ and a $\overline{u}$ such that:

(1) $\Gamma \vdash t :\longrightarrow^*: \overline{t} : \Pi\, F\, G$
(2) $\Gamma \vdash u :\longrightarrow^*: \overline{u} : \Pi\, F\, G$
(3) $\Gamma \vdash \overline{t} \cong \overline{u} : \Pi\, F\, G$
(4) $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$
(5) $\Gamma \Vdash_\ell u : A/_{\mathcal{T}}$
(6) $\forall \rho : \Delta \le \Gamma,\ u :: \Delta \Vdash_\ell a : F[\rho]/_{\mathcal{U}(\rho)}.\ \ \Delta \Vdash_\ell \overline{t}[\rho]\, a = \overline{u}[\rho]\, a : G[\rho, a]/_{\mathcal{V}(\rho,u)}$

In requirement 6, we say that under weakening ρ, given a term a of type $F[\rho]$, $\overline{t}[\rho]$ and $\overline{u}[\rho]$ are equal under application of a. This is necessary to prove congruence of application.

Embedding. Reducible small types can also be viewed as reducible large types.

$$\frac{\mathcal{U} :: \Gamma \Vdash_{\ell'} A}{\Gamma \Vdash_\ell A}\ \ell' < \ell$$

Given a derivation $\mathcal{T}$ built by that rule:

- $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ iff $\Gamma \Vdash_{\ell'} A = B/_{\mathcal{U}}$.
- $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ iff $\Gamma \Vdash_{\ell'} t : A/_{\mathcal{U}}$. *This allows type level embedding of reducible terms.*
- $\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}$ iff $\Gamma \Vdash_{\ell'} t = u : A/_{\mathcal{U}}$.

3.3 Properties of the Logical Relation

In this section, we establish some basic properties of the logical relation. We start by reflexivity, showing that reducible objects are well-formed, and logically related objects are generically equal.

Lemma 3.1 (Reflexivity). *Given* $\mathcal{T} :: \Gamma \Vdash_\ell A$ *then:*

(1) $\Gamma \Vdash_\ell A = A/_{\mathcal{T}}$.
(2) $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ *then* $\Gamma \Vdash_\ell t = t : A/_{\mathcal{T}}$.

Proof. By induction on $\mathcal{T}$. □

Lemma 3.2 (Escape lemma). *Given* $\mathcal{T} :: \Gamma \Vdash_\ell A$ *then:*

(1) $\Gamma \vdash A$.
(2) *If* $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ *then* $\Gamma \vdash A \cong B$.
(3) *If* $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ *then* $\Gamma \vdash t : A$.
(4) *If* $\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}$ *then* $\Gamma \vdash t \cong u : A$.

Proof. By induction on $\mathcal{T}$ and reduction being subsumed by equality (2.2). □

For much of the following proofs, we will need to use induction on two or more derivations of type reducibility for types that are reducibly equal. As there are four different cases for typing derivations (ignoring the embedding case, as one can immediately recurse on its premise), with two derivations we would have 16 cases and with three derivation we would have 64 cases. However, it turns out that a lot of cases can be refuted by the reduction properties. We would like to avoid repeating this refutation process and instead only prove it once.

We introduce what we call the shape view, which is an inductively defined relation on two or more type reducibility derivations such that an instance of the view is only valid if the derivations have a compatible inductive structure, i.e. either they are built with the same inference rule or one of them is an embedding, for which we use the structure of the premise. For example, if we have two type reducibility derivations $\mathcal{T}$ and $\mathcal{U}$, and $\mathcal{T}$ is built using the natural number type rule:

$$\frac{\Gamma \vdash A :\longrightarrow^* : \mathbb{N}}{\Gamma \Vdash_\ell A}$$

Then for $\mathcal{U}$ to be related by the view with $\mathcal{T}$, it must be built by the same rule either directly or inside the embedding case. Pattern matching on proofs of the view then allows us to consider only the compatible cases and not have to worry everywhere about the ones where e.g. both $A \longrightarrow^* \Pi\, F\, G$ and $A \longrightarrow^* \mathbb{N}$.

Lemma 3.3 (Shape view construction). *Given* $\mathcal{T} :: \Gamma \Vdash_\ell A$ *and* $\mathcal{T}' :: \Gamma \Vdash_{\ell'} B$, *if* $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ *then there is an shape view of* $\mathcal{T}$ *and* $\mathcal{T}'$.

Proof. By induction on $\mathcal{T}$ and $\mathcal{T}'$, as Whnfs do not reduce (2.4) and reduction is deterministic (2.5). □

Corollary 3.4 (Reflexive shape view construction). *Given* $\mathcal{T} :: \Gamma \Vdash_\ell A$ *and* $\mathcal{T}' :: \Gamma \Vdash_{\ell'} A$, *there is an shape view of* $\mathcal{T}$ *and* $\mathcal{T}'$.

Proof. Directly by reflexivity of reducible equality (3.1) and shape view construction (3.3). □

Lemma 3.5 (Irrelevance). *Given* $\mathcal{T} :: \Gamma \Vdash_\ell A$ *and* $\mathcal{T}' :: \Gamma \Vdash_{\ell'} A$ *then:*

(1) *If* $\Gamma \Vdash_\ell A = B/_{\mathcal{T}}$ *then* $\Gamma \Vdash_{\ell'} A = B/_{\mathcal{T}'}$.
(2) *If* $\Gamma \Vdash_\ell t : A/_{\mathcal{T}}$ *then* $\Gamma \Vdash_{\ell'} t : A/_{\mathcal{T}'}$.
(3) *If* $\Gamma \Vdash_\ell t = u : A/_{\mathcal{T}}$ *then* $\Gamma \Vdash_{\ell'} t = u : A/_{\mathcal{T}'}$.

Proof. By induction on the shape view of $\mathcal{T}$ and $\mathcal{T}'$ given by reflexive shape view construction (3.4) and by determinism of reduction (2.5). □

Even if $\mathcal{T}$ and $\mathcal{T}'$ above can actually differ, e.g. by containing different typing derivations, our irrelevance lemma shows that for the recursively defined judgements, the specific proof of type reducibility $\mathcal{T}$ does not matter. We can therefore use these judgements more freely as we no longer need to refer to a specific type reducibility derivation.

Based on this intuition, we will from here on drop the type derivation from the reducibility judgements and instead implicitly state that there exists such a derivation for a judgement. More formally, we let $\boxed{\Gamma \Vdash_\ell J : A}$ mean that $\Gamma \Vdash_\ell J : A/_{\mathcal{T}}$ holds for some $\mathcal{T} :: \Gamma \Vdash_\ell A$. However, this definition is not practical in the implementation as often judgements will share the same type, and thus using only the above definition would introduce extra complications. Therefore, in the implementation, we use derivations explicitly.

Lemma 3.6 (Weakening). *For all judgements J of the logical relation, given $\rho : \Delta \leq \Gamma$ and $\Gamma \Vdash_\ell J$ then $\Delta \Vdash_\ell J[\rho]$.*

Proof. By induction on the derivation of type A and by weakening of well-formed expressions (2.7) and irrelevance (3.5). □

Lemma 3.7 (Conversion). *Given $\Gamma \Vdash_\ell A :=: B$ then:*

(1) *$\Gamma \Vdash_\ell t : A$ iff $\Gamma \Vdash_\ell t : B$.*
(2) *$\Gamma \Vdash_\ell t = u : A$ iff $\Gamma \Vdash_\ell t = u : B$.*

Proof. By induction on the shape view (3.3) of equal types A and B, using determinism of reduction (2.5) and irrelevance 3.5. The two directions of iff are proven simultaneously. □

Lemma 3.8 (Symmetry).

(1) *If $\Gamma \Vdash_\ell A :=: B$ then $\Gamma \Vdash_\ell B = A$.*
(2) *If $\Gamma \Vdash_\ell t = u : A$ then $\Gamma \Vdash_\ell u = t : A$.*

Proof. By induction on the shape view (3.3) of the equal types A and B, using determinism of reduction (2.5), irrelevance (3.5) and conversion for reducible equality (3.7). □

Lemma 3.9 (Transitivity).

(1) *If $\Gamma \Vdash_\ell A :=: A'$ and $\Gamma \Vdash_\ell A' :=: A''$ then $\Gamma \Vdash_\ell A = A''$.*
(2) *If $\Gamma \Vdash_\ell t = t' : A$ and $\Gamma \Vdash_\ell t' = t'' : A$ then $\Gamma \Vdash_\ell t = t'' : A$.*

Proof. By induction on the shape view (3.3) of type A, A' and A'', using determinism of reduction (2.5), irrelevance (3.5) and conversion of reducible equality (3.7). □

Lemma 3.10 (Neutrals are reducible). *Let $\Gamma \Vdash_\ell A$ and $\Gamma \vdash n : A$.*

(1) *If $\Gamma \vdash n :\sim: n : A$ then $\Gamma \Vdash_\ell n : A$.*
(2) *If $\Gamma \vdash n' : A$ and $\Gamma \vdash n :\sim: n' : A$ then $\Gamma \Vdash_\ell n = n' : A$.*

Proof. By induction on the derivation of type A and by well-formed weakening (2.7), reduction is subsumed by equality (2.2) and escape (3.2). □

Lemma 3.11 (Weak head expansion).

(1) *If $\Gamma \Vdash_\ell B$ and $\Gamma \vdash A \longrightarrow^* B$ then $\Gamma \Vdash_\ell A :=: B$.*
(2) *If $\Gamma \Vdash_\ell u : B$ and $\Gamma \vdash t \longrightarrow^* u : B$ then $\Gamma \Vdash_\ell t :=: u : B$.*

Proof. By induction on the reducibility of B, subject typing (2.3) and reflexivity (3.1). □

Lemma 3.12 (Application reducibility). *Given* $\Gamma \Vdash_\ell \Pi\, F\, G$ *and* $\Gamma \Vdash_\ell u : F$ *and* $\Gamma \Vdash_\ell G[u]$ *then:*
(1) *If* $\Gamma \Vdash_\ell t : \Pi\, F\, G$ *then* $\Gamma \Vdash_\ell t\, u : G[u]$.
(2) *If* $\Gamma \Vdash_\ell t = t' : \Pi\, F\, G$ *and* $\Gamma \Vdash_\ell u :=: u' : F$ *then* $\Gamma \Vdash_\ell t\, u = t'\, u' : G[u]$.

Proof. For case (1): By applying u instead of a in the following premise of the reducible term t:

$$\forall \rho : \Delta \le \Gamma.\ \ \Delta \Vdash_\ell a : F[\rho] \implies \Delta \Vdash_\ell \bar{t}[\rho]\, a : G[\rho, a]$$

Case (2) is solved similarly. This proof is accomplished with escape (3.2), irrelevance (3.5), conversion (3.7), symmetry (3.8), transitivity (3.9) and weak head expansion (3.11). □

3.4 Validity Judgements

In Section 3.2, we have introduced reducible objects $\Gamma \Vdash_\ell t : A$, which are guaranteed to be well-typed, $\Gamma \vdash t : A$, but we cannot prove the converse yet, i.e., that each well-typed term is reducible. In other words, reducible objects do not directly provide a model of $\lambda^{\Pi\mathbb{U}\mathbb{N}}$, in the sense that the interpretation function cannot be implemented by a naive induction on the derivation. The simplest counterexample is the typing rule for λ-abstraction:

$$\frac{\Gamma, F \vdash t : G}{\Gamma \vdash \lambda t : \Pi\, F\, G}$$

By definition of $\Gamma \Vdash_\ell \lambda t : \Pi\, F\, G$, we have to show, amongst others, that for any $\rho : \Delta \le \Gamma$ and any $\Delta \Vdash_\ell a : F$, we have $\Delta \Vdash_\ell (\lambda t)[\rho]\, a : G[\rho, a]$. Using weak head expansion, it is sufficient to show $\Delta \Vdash_\ell t[\rho, a] : G[\rho, a]$. However, our induction hypothesis $\Gamma, F \Vdash_\ell t : G$ is too weak to prove that. We would need to substitute with $\Delta \vdash (\rho, a) : (\Gamma, F)$, but reducible terms are not closed under substitution. The way out is by blunt force: we define *valid* objects $\Gamma \Vdash^{\mathrm{v}}_\ell t : A$ to be objects that are reducible under substitution with reducible objects. Those then model our syntax, i.e., we can prove the fundamental theorem (3.21) which states that $\Gamma \vdash t : A$ implies $\Gamma \Vdash^{\mathrm{v}}_\ell t : A$. Applying valid object t to the identity substitution (3.17), we obtain reducibility $\Gamma \Vdash_\ell t : A$ and all the good properties that follow from it.

For each of the reducibility judgments we then define their validity counterparts. We will make use of judgments for validity of contexts $\mathcal{G} :: \Vdash^{\mathrm{v}} \Gamma$, reducible substitutions $\mathcal{S} :: \Delta \Vdash^{\mathrm{s}} \sigma : \Gamma/_{\mathcal{G}}$ and their equality $\Delta \Vdash^{\mathrm{s}} \sigma = \sigma' : \Gamma/_{\mathcal{G}}/_{\mathcal{S}}$ which we will define just after through induction-recursion.

First we define valid types as those that are reducible under any reducible substitution and respect their equality. $\boxed{\Gamma \Vdash^{\mathrm{v}}_\ell A/_{\mathcal{G}}}$ iff for all $\mathcal{S} :: \Delta \Vdash^{\mathrm{s}} \sigma : \Gamma/_{\mathcal{G}}$ it holds that:

(1) $\mathcal{U} :: \Delta \Vdash_\ell A[\sigma]$
(2) $\Delta \Vdash^{\mathrm{s}} \sigma' : \Gamma/_{\mathcal{G}}$ and $\Delta \Vdash^{\mathrm{s}} \sigma = \sigma' : \Gamma/_{\mathcal{G}}/_{\mathcal{S}}$ imply $\Delta \Vdash_\ell A[\sigma] = A[\sigma']/_{\mathcal{U}}$

Note that we will use the above enumeration indexes as projections, e.g., given $\mathcal{T} :: \Gamma \Vdash^{\mathrm{v}}_\ell A/_{\mathcal{G}}$ the notation $\mathcal{T}(\mathcal{S}).1$ stands for the proof of that $A[\sigma]$ is reducible.

Given $\mathcal{T}$ as above we define valid type equality, valid terms and valid term equality:
$\boxed{\Gamma \Vdash^{\mathrm{v}}_\ell A = B/_{\mathcal{G}}/_{\mathcal{T}}}$ iff for all $\mathcal{S} :: \Delta \Vdash^{\mathrm{s}} \sigma : \Gamma/_{\mathcal{G}}$ we have $\Delta \Vdash_\ell A[\sigma] = B[\sigma]/_{\mathcal{T}(\mathcal{S}).1}$.

$\boxed{\Gamma \Vdash^{\mathrm{v}}_\ell t : A/_{\mathcal{G}}/_{\mathcal{T}}}$ iff for all $\mathcal{S} :: \Delta \Vdash^{\mathrm{s}} \sigma : \Gamma/_{\mathcal{G}}$ it holds that:

(1) $\Delta \Vdash_\ell t[\sigma] : A[\sigma]/_{\mathcal{T}(\mathcal{S}).1}$
(2) $\Delta \Vdash^{\mathrm{s}} \sigma' : \Gamma/_{\mathcal{G}}$ and $\Delta \Vdash^{\mathrm{s}} \sigma = \sigma' : \Gamma/_{\mathcal{G}}/_{\mathcal{S}}$ imply $\Delta \Vdash_\ell t[\sigma] = t[\sigma'] : A[\sigma]/_{\mathcal{T}(\mathcal{S}).1}$

$\boxed{\Gamma \Vdash^{\mathrm{v}}_\ell t = u : A/_{\mathcal{G}}/_{\mathcal{T}}}$ iff for all $\mathcal{S} :: \Delta \Vdash^{\mathrm{s}} \sigma : \Gamma/_{\mathcal{G}}$ we have $\Delta \Vdash_\ell t[\sigma] = u[\sigma] : A[\sigma]/_{\mathcal{T}(\mathcal{S}).1}$.

By induction-recursion, we define $\boxed{\Vdash^{\mathrm{v}} \Gamma}$ inductively by two rules, one for the empty context and one for context expansion, and simultaneously define $\boxed{\Delta \Vdash^{\mathrm{s}} \sigma : \Gamma/_{\mathcal{G}}}$ and $\boxed{\Delta \Vdash^{\mathrm{s}} \sigma = \sigma' : \Gamma/_{\mathcal{G}}/_{\mathcal{S}}}$ by

recursion on the derivation $\mathcal{G} :: \Vdash^{\mathsf{v}} \Gamma$, the second depending on $\mathcal{S} :: \Delta \Vdash^{\mathsf{s}} \sigma : \Gamma/_{\mathcal{G}}$. For the following definitions, let Δ be a well-formed context.

Empty Context.

$$\frac{}{\Vdash^{\mathsf{v}} \epsilon}$$

For derivations $\mathcal{G}$ built with that rule we define that $\Delta \Vdash^{\mathsf{s}} \sigma : \epsilon/_{\mathcal{G}}$ holds unconditionally and for all $\mathcal{S} :: \Delta \Vdash^{\mathsf{s}} \sigma : \epsilon/_{\mathcal{G}}$ we have $\Delta \Vdash^{\mathsf{s}} \sigma = \sigma' : \epsilon/_{\mathcal{G}}/_{\mathcal{S}}$.

Extended Context.

$$\frac{\mathcal{G}' :: \Vdash^{\mathsf{v}} \Gamma \qquad \mathcal{T} :: \Gamma \Vdash^{\mathsf{v}}_{\ell} A/_{\mathcal{G}'}}{\Vdash^{\mathsf{v}} \Gamma, A}$$

For derivations $\mathcal{G}$ built with that rule we define:

- $\Delta \Vdash^{\mathsf{s}} \sigma : \Gamma, A/_{\mathcal{G}}$ iff
  (1) $\mathcal{S}' :: \Delta \Vdash^{\mathsf{s}} \sigma \circ \uparrow\mathsf{id} : \Gamma/_{\mathcal{G}'}$ *meaning that the tail of σ is reducible.*
  (2) $\Delta \Vdash_{\ell} i_0[\sigma] : A[\sigma \circ \uparrow\mathsf{id}]/_{\mathcal{T}(\mathcal{S}').1}$ *meaning that the head of σ is reducible.*
- For all $\mathcal{S} :: \Delta \Vdash^{\mathsf{s}} \sigma : \Gamma, A/_{\mathcal{G}}$ we have $\Delta \Vdash^{\mathsf{s}} \sigma = \sigma' : \Gamma, A/_{\mathcal{G}}/_{\mathcal{S}}$ iff
  (1) $\Delta \Vdash^{\mathsf{s}} \sigma \circ \uparrow\mathsf{id} = \sigma' \circ \uparrow\mathsf{id} : \Gamma/_{\mathcal{G}'}/_{\mathcal{S}.1}$
  (2) $\Delta \Vdash_{\ell} i_0[\sigma] = i_0[\sigma'] : A[\sigma \circ \uparrow\mathsf{id}]/_{\mathcal{T}(\mathcal{S}.1)}$
  Note that $\mathcal{S}.1$ stands for the proof that the tail of σ is reducible.

3.5 Properties of the Validity Judgements

In preparation for the fundamental theorem, we prove a few properties about validity and reducible substitutions.

As the validity judgements are defined using induction-recursion, we will—similarly to the logical relation—prove context and typing derivation irrelevance.

Lemma 3.13 (Irrelevance). *Let $\mathcal{G} :: \Vdash^{\mathsf{v}} \Gamma$ and $\mathcal{G}' :: \Vdash^{\mathsf{v}} \Gamma$.*

(1) *If $\mathcal{S} :: \Delta \Vdash^{\mathsf{s}} \sigma : \Gamma/_{\mathcal{G}}$ then there is a derivation $\mathcal{S}' :: \Delta \Vdash^{\mathsf{s}} \sigma : \Gamma/_{\mathcal{G}'}$.*
If further $\Delta \Vdash^{\mathsf{s}} \sigma = \sigma' : \Gamma/_{\mathcal{G}}/_{\mathcal{S}}$ then $\Delta \Vdash^{\mathsf{s}} \sigma = \sigma' : \Gamma/_{\mathcal{G}'}/_{\mathcal{S}'}$.
(2) *If $\mathcal{T} :: \Gamma \Vdash^{\mathsf{v}}_{\ell} A/_{\mathcal{G}}$ then there is a derivation $\mathcal{T}' :: \Gamma \Vdash^{\mathsf{v}}_{\ell} A/_{\mathcal{G}'}$. It also follows that:*
  (a) *If $\Gamma \Vdash^{\mathsf{v}}_{\ell} A = B/_{\mathcal{G}}/_{\mathcal{T}}$ then $\Gamma \Vdash^{\mathsf{v}}_{\ell} A = B/_{\mathcal{G}'}/_{\mathcal{T}'}$.*
  (b) *If $\Gamma \Vdash^{\mathsf{v}}_{\ell} t : A/_{\mathcal{G}}/_{\mathcal{T}}$ then $\Gamma \Vdash^{\mathsf{v}}_{\ell} t : A/_{\mathcal{G}'}/_{\mathcal{T}'}$.*
  (c) *If $\Gamma \Vdash^{\mathsf{v}}_{\ell} t = u : A/_{\mathcal{G}}/_{\mathcal{T}}$ then $\Gamma \Vdash^{\mathsf{v}}_{\ell} t = u : A/_{\mathcal{G}'}/_{\mathcal{T}'}$.*

Proof. By induction on $\mathcal{G}$ and $\mathcal{G}'$, using irrelevance for the reducibility judgements (3.5). □

Derivation irrelevance justifies that we from here on drop the context and type derivation arguments from the validity judgements, as we do for reducibility. Further, when we state $\Delta \Vdash^{\mathsf{s}} \sigma : \Gamma$ or $\Delta \Vdash^{\mathsf{s}} \sigma = \sigma' : \Gamma$ we presuppose $\Vdash^{\mathsf{v}} \Gamma$.

The escape lemma extends to reducible substitutions:

Lemma 3.14 (Escape for substitutions).

(1) *If $\Delta \Vdash^{\mathsf{s}} \sigma : \Gamma$ then $\Delta \vdash \sigma : \Gamma$.*
(2) *If $\Delta \Vdash^{\mathsf{s}} \sigma = \sigma' : \Gamma$ then $\Delta \vdash \sigma = \sigma' : \Gamma$.*

Proof. By induction on $\Vdash^{\mathsf{v}} \Gamma$, using the escape lemma (3.2). □

The following lemma is needed to make the fundamental theorem go through the binder cases (λ, Π), in particular, to establish the well-formedness and well-typedness conditions there. It relies

on weakening for the reducibility judgements, and is the reason why our logical relation is *Kripke* in the first place.

Lemma 3.15 (Substitution weakening). *Given* $\rho : \Delta' \leq \Delta$ *and* $\Delta \Vdash^{s} \sigma : \Gamma$ *then* $\Delta' \Vdash^{s} \rho \circ \sigma : \Gamma$. *If further* $\Delta \Vdash^{s} \sigma = \sigma' : \Gamma$ *then* $\Delta' \Vdash^{s} \rho \circ \sigma = \rho \circ \sigma' : \Gamma$.

Proof. By induction on $\Vdash^{v} \Gamma$, using weakening for the reducibility judgements (3.6). □

Lemma 3.16 (Substitution lifting). *Given* $\Gamma \Vdash^{v}_{\ell} A/_{\mathcal{E}}$ *and* $\Delta \Vdash^{s} \sigma : \Gamma$ *then* $\Delta, A[\sigma] \Vdash^{s} \Uparrow\sigma : \Gamma, A$. *If further* $\Delta \Vdash^{s} \sigma = \sigma' : \Gamma$ *then* $\Delta, A[\sigma] \Vdash^{s} \Uparrow\sigma = \Uparrow\sigma' : \Gamma, A$.

Proof. By reducibility of neutrals (3.10) and substitution weakening (3.15). □

Lemma 3.17 (Identity substitution). *If* $\Vdash^{v} \Gamma$ *then* $\vdash \Gamma$ *and* $\Gamma \Vdash^{s} \mathsf{id} : \Gamma$.

Proof. By induction on the validity of Γ, escape (3.2), irrelevance (3.5) and substitution lifting (3.16). □

Lemma 3.18 (Substitution equivalence). $\Delta \Vdash^{s} \_ = \_ : \Gamma$ *is an equivalence relation on valid substitutions.*

(1) *If* $\Delta \Vdash^{s} \sigma : \Gamma$ *then* $\Delta \Vdash^{s} \sigma = \sigma : \Gamma$.
(2) *If* $\Delta \Vdash^{s} \sigma :=: \sigma' : \Gamma$ *then* $\Delta \Vdash^{s} \sigma' = \sigma : \Gamma$.
(3) *If* $\Delta \Vdash^{s} \sigma :=: \sigma' : \Gamma$ *and* $\Delta \Vdash^{s} \sigma' :=: \sigma'' : \Gamma$ *then* $\Delta \Vdash^{s} \sigma = \sigma'' : \Gamma$.

Proof. By induction on $\Vdash^{v} \Gamma$, using reflexivity (3.1), conversion (3.7), symmetry (3.8) and transitivity (3.9) of reducible equality. □

Corollary 3.19 (Reducibility of validity). *All valid types, terms and equalities are reducible.*

Proof. By applying the reducible identity substitution (3.17) to the valid object and using reducible irrelevance (3.5). □

While we can substitute valid objects with arbitrary reducible substitutions and get reducible objects, we do not directly get valid objects. This is because reducibility is not closed under substitution. Thus we will restrict ourselves to proving that substitution of a single term preserves validity.

Lemma 3.20 (Single substitution). *Given* $\Gamma \Vdash^{v}_{\ell} F$:

(1) *If* $\Gamma \Vdash^{v}_{\ell} t : F$ *and either* $\Gamma, F \Vdash^{v}_{\ell} G$ *or* $\Gamma \Vdash^{v}_{\ell} \Pi\, F\, G$ *then* $\Gamma \Vdash^{v}_{\ell} G[t]$.
(2) *If* $\Gamma \Vdash^{v}_{\ell} F :=: F'$ *and* $\Gamma \Vdash^{v}_{\ell} t :=: t' : F$ *with* $\Gamma \Vdash^{v}_{\ell} t' : F'$ *and either* $\Gamma, F \Vdash^{v}_{\ell} G :=: G'$ *with* $\Gamma, F' \Vdash^{v}_{\ell} G'$ *or* $\Gamma \Vdash^{v}_{\ell} \Pi\, F\, G :=: \Pi\, F'\, G'$ *then* $\Gamma \Vdash^{v}_{\ell} G[t] = G'[t']$.
(3) *If* $\Gamma \Vdash^{v}_{\ell} t : F$ *and* $\Gamma, F \Vdash^{v}_{\ell} G$ *and* $\Gamma, F \Vdash^{v}_{\ell} f : G$ *then* $\Gamma \Vdash^{v}_{\ell} f[t] : G[t]$.
(4) *If* $\Gamma, F \Vdash^{v}_{\ell} u : F[\uparrow\mathsf{id}]$ *then* $\Gamma, F \Vdash^{v}_{\ell} G[\uparrow\mathsf{id}, u]$.
(5) *If* $\Gamma, F \Vdash^{v}_{\ell} u :=: u' : F[\uparrow\mathsf{id}]$ *then* $\Gamma, F \Vdash^{v}_{\ell} G[\uparrow\mathsf{id}, u] = G'[\uparrow\mathsf{id}, u']$.

Proof. We elaborate case (1): If $\Gamma, F \Vdash^{v}_{\ell} G$, we deconstruct that validity judgement and reconstruct it such that when applying a substitution σ, we instead apply $(\sigma, t[\sigma])$. Otherwise, if $\Gamma \Vdash^{v}_{\ell} \Pi\, F\, G$, we deconstruct the reducible object of the validity judgement and replace a with $t[\sigma]$ in the following premise of reducible Π:

$$\forall \rho : \Delta \leq \Gamma.\ \Delta \Vdash_{\ell} a : F[\rho] \implies \Delta \Vdash_{\ell} G[\rho, a]$$

All in all, to prove this lemma we need escape (3.2), irrelevance (3.5), conversion (3.7) and transitivity (3.9) of reducible equality, and reflexivity of substitution (3.18). □

With the above lemmas, we can now prove the fundamental theorem:

Theorem 3.21 (Fundamental theorem).
(1) *If* $\vdash \Gamma$ *then* $\Vdash^{\mathrm{v}} \Gamma$.
(2) *If* $\Gamma \vdash A$ *then* $\Gamma \Vdash^{\mathrm{v}}_{\ell} A$.
(3) *If* $\Gamma \vdash A = B$ *then* $\Gamma \Vdash^{\mathrm{v}}_{\ell} A :=: B$.
(4) *If* $\Gamma \vdash t : A$ *then* $\Gamma \Vdash^{\mathrm{v}}_{\ell} A$ *and* $\Gamma \Vdash^{\mathrm{v}}_{\ell} t : A$.
(5) *If* $\Gamma \vdash t = u : A$ *then* $\Gamma \Vdash^{\mathrm{v}}_{\ell} A$ *and* $\Gamma \Vdash^{\mathrm{v}}_{\ell} t :=: u : A$.

Proof. By induction on the well-formed judgements. The most salient cases are, Π and Π-congruence, variables, λ and η-equality, application and application-congruence, and finally natrec and natrec-congruence.

This is accomplished with escape (3.2), reducible equality being a equivalence relation (3.1 & 3.8 & 3.9), conversion (3.7), irrelevance of reducibility (3.5) and validity (3.13), reducibility of neutrals (3.10), weak head expansion (3.11), reducibility of application (3.12), substitution weakening (3.15) and lifting (3.16), reflexivity of substitutions (3.18) and single substitution (3.20). □

With the fundamental theorem, escape lemma (3.2) and reducibility of validity (3.19) we have effectively proven $\Gamma \vdash J$ iff $\Gamma \Vdash^{\mathrm{v}} J$ for the judgements J for types, terms and their respective equality. From this we can also prove $\Gamma \vdash J$ iff $\Gamma \Vdash J$:

Corollary 3.22 (Reducibility of well-formedness). *Any well-formed object is reducible.*

Proof. Well-formedness implies validity by the fundamental theorem (3.21) instantiated to judgemental equality (1). Further, validity implies reducibility (3.19). □

We will also prove a fundamental theorem for substitutions:

Theorem 3.23 (Fundamental theorem for substitutions). *If* $\Delta \vdash \sigma : \Gamma$ *for well-formed contexts* Γ *and* Δ *then* $\Delta \Vdash^{\mathrm{s}} \sigma : \Gamma$.

Proof. By induction on Γ and with the judgmental instance (1): by fundamental theorem (3.21) and reducible irrelevance (3.5), valid irrelevance (3.13) and identity substitution (3.17). □

3.6 Consequences of the Fundamental Theorem

We will here declare and prove some theorems we can now prove using the fundamental theorem with generic equality instance 1.

Theorem 3.24 (Canonicity). *Let* $\mathrm{suc}^0\, t \triangleq t$ *and* $\mathrm{suc}^{n+1}\, t \triangleq \mathrm{suc}\,(\mathrm{suc}^n\, t)$. *Given* $\epsilon \vdash t : \mathbb{N}$ *then there exists* n *such that* $\epsilon \vdash t = \mathrm{suc}^n\, \mathrm{zero} : \mathbb{N}$.

Proof. Since well-formed objects are reducible (3.22) we have $\epsilon \Vdash \mathbb{N}\ t$, on which we induct. □

Theorem 3.25 (Π-injectivity). *If* $\Gamma \vdash \Pi\, F\, G = \Pi\, H\, E$ *then* $\Gamma \vdash F = H$ *and* $\Gamma, F \vdash G = E$.

Proof. By well-formed objects being reducible (3.22) we have $\Gamma \Vdash_{\ell} \Pi\, F\, G = \Pi\, H\, E$, from which injectivity can be retrieved by escape (3.2), irrelevance (3.5) and reducibility of neutrals (3.10). □

Theorem 3.26 (Syntactic validity).
(1) *If* $\Gamma \vdash A = B$ *then* $\Gamma \vdash A$ *and* $\Gamma \vdash B$.
(2) *If* $\Gamma \vdash t : A$ *then* $\Gamma \vdash A$.
(3) *If* $\Gamma \vdash t = u : A$ *then* $\Gamma \vdash t : A$ *and* $\Gamma \vdash u : A$.
(4) *If* $\Gamma \vdash A \longrightarrow^* B$ *then* $\Gamma \vdash A$ *and* $\Gamma \vdash B$.
(5) *If* $\Gamma \vdash t \longrightarrow^* u : A$ *then* $\Gamma \vdash t : A$ *and* $\Gamma \vdash u : A$.

Proof. By the fundamental theorem (3.21) and by reductions being subsumed by equality (2.2) and escape (3.2). □

Corollary 3.27 (Syntactic Pi). *If* $\Gamma \vdash \Pi\, F\, G$ *then* $\Gamma \vdash F$ *and* $\Gamma, F \vdash G$.

Proof. By lemma Π-injectivity (3.25) and syntactic validity (3.26). □

Theorem 3.28 (Weak head normalization).

(1) *If* $\Gamma \vdash A$ *then there exists some* $\overline{A}$ *such that* $\Gamma \vdash A \longrightarrow^* \overline{A}$.
(2) *If* $\Gamma \vdash t : A$ *then there exists some* $\overline{t}$ *such that* $\Gamma \vdash t \longrightarrow^* \overline{t} : A$.

Proof. By well-formed objects being reducible (3.22) we get a reducible object for which we use induction and reduction being subsumed by equality (2.2). □

Theorem 3.29 (Syntactic equality).

(1) *If* $\Gamma \vdash \mathsf{U} = A$ *then* $\mathsf{U} \equiv A$.
(2) *If* $\Gamma \vdash \mathbb{N} = \overline{A}$ *then* $\mathbb{N} \equiv \overline{A}$.
(3) *If* $\Gamma \vdash N = \overline{A}$ *then there exists* M *such that* $M \equiv \overline{A}$.
(4) *If* $\Gamma \vdash \Pi\, F\, G = \overline{A}$ *then there exist* H *and* E *such that* $\Pi\, H\, E \equiv \overline{A}$.

Proof. By well-formed objects being reducible (3.22) and induction on the reducible equality with reducible irrelevance (3.5). □

Theorem 3.30 (Syntactic inequality). *For* $A, B \in \{\mathsf{U}, \mathbb{N}, N, \Pi\, F\, G\}$, *if* $A \not\equiv B$ *then* $\Gamma \vdash A \neq B$.

Proof. By well-formed objects being reducible (3.22) and showing that there cannot exist an instance of the shape view of types A and B using escape (3.2), shape view construction (3.3) and reducible irrelevance (3.5). □

Theorem 3.31 (Substitution).

(1) *If* $\Delta \vdash \sigma : \Gamma$ *and* $\Gamma \vdash A$ *then* $\Delta \vdash A[\sigma]$.
(2) *If* $\Delta \vdash \sigma = \sigma' : \Gamma$ *and* $\Gamma \vdash A = B$ *then* $\Delta \vdash A[\sigma] = B[\sigma']$.
(3) *If* $\Delta \vdash \sigma : \Gamma$ *and* $\Gamma \vdash t : A$ *then* $\Delta \vdash t[\sigma] : A[\sigma]$.
(4) *If* $\Delta \vdash \sigma = \sigma' : \Gamma$ *and* $\Gamma \vdash t = u : A$ *then* $\Delta \vdash t[\sigma] = u[\sigma'] : A[\sigma]$.

Proof. By the fundamental theorem (3.21) and fundamental theorem for substitutions (3.23), escape (3.2) and valid irrelevance (3.13). □

Theorem 3.32 (Substitution composition). *Given* $\Delta' \vdash \sigma : \Delta$ *and* $\Delta \vdash \sigma' : \Gamma$ *then* $\Delta' \vdash \sigma \circ \sigma' : \Gamma$

Proof. By induction on the well-formedness of Γ and σ, using the substitution theorem(3.31). □

Theorem 3.33 (Neutral type equality). *If* $\Gamma \vdash n : A$ *and* $\Gamma \vdash n : B$ *then* $\Gamma \vdash A = B$.

Proof. By induction on the syntactic structure of n and by Π-injectivity (3.25), syntactic validity (3.26) and substitution (3.31). □

Theorem 3.34 (Universe membership).

(1) *If* $\Gamma \vdash A$ *and* A *has no occurrence of* U *then* $\Gamma \vdash A : \mathsf{U}$.
(2) *If* $\Gamma \vdash A = B$ *and* A *and* B *has no occurrence of* U *then* $\Gamma \vdash A = B : \mathsf{U}$.

Proof. By induction on the judgements and by syntactic validity (3.26). □

Theorem 3.35 (Consistency). $\Gamma \vdash \mathsf{zero} = \mathsf{suc}\ \mathsf{zero} : \mathbb{N}$ *is impossible.*

Proof. By well-formed objects being reducible (3.22), using induction on the reducible instance of the equality and using the fact that Whnf do not reduce (2.4) and reducible irrelevance (3.5). □

4 DECIDABILITY

To prove that our language has decidable conversion, we will now introduce an algorithm for conversion of types and terms. The algorithm is defined as a relation which we then show decidable and equivalent to the conversion judgements. In particular completeness is proven by constructing a generic equality instance for the algorithmic equality, so that we can apply our fundamental theorem.

4.1 Conversion Algorithm

Our conversion algorithm is defined inductively as seen in Fig. 6, with six different relations defined simultaneously.

We first have the relations $\Gamma \vdash n \longrightarrow m : A$ and $\Gamma \vdash n \mathrel{\hat{\longrightarrow}} m : A$, which is the algorithm that determines equality between neutral terms, where the former relation enforces the type A to be in Whnf. Secondly we have the relations $\Gamma \vdash A \Longleftrightarrow B$ and $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$, which determines equality between types, where the former of the two relations enforces the types to be in Whnf. Lastly, we have the relations $\Gamma \vdash t \Longleftrightarrow u : A$ and $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$, which determines equality between terms. Similarly to above the former enforces Whnf of the type and the two terms.

Of note is the third rule of these two relations. We do not check that the type N is equal to the type M inferred by the neutral algorithm. Since we can derive this equality by Lemma 3.33, we can be economic and drop this premise, thus simplifying the relation.

4.2 Properties of the Conversion Algorithm

Our next goal is to prove decidability and construct a generic equality instance for the logical relation. It turns out that some of the properties necessary for instance validity is also necessary for proving decidability, thus we will begin proving those properties.

To prove these properties, we also need a notion of context equality, which we will denote as $\boxed{\vdash \Gamma = \Delta}$. We define it inductively:

$$\frac{}{\vdash \epsilon = \epsilon} \qquad \frac{\vdash \Gamma = \Delta \qquad \Gamma \vdash A = B}{\vdash \Gamma, A = \Delta, B}$$

Lemma 4.1 (Context conversion for typing judgements). *Given* $\vdash \Gamma = \Delta$*:*

(1) $\Delta \vdash \mathrm{id} : \Gamma$ *and contexts* Γ *and* Δ *are well-formed.*
(2) *If* $\Gamma \vdash J$ *then* $\Delta \vdash J$ *for the syntactic judgements* J *of types, type membership and their respective equality.*

Proof. By induction on the context equality and by substitution (3.31). □

Lemma 4.2 (Context conversion for reduction and algorithmic equality). *If* $\vdash \Gamma = \Delta$ *and* $\Gamma \vdash J$ *then* $\Delta \vdash J$ *for the syntactic judgements* J *of reduction and algorithmic equality.*

Proof. By induction on the judgements and context conversion for typing judgements (4.1). □

Lemma 4.3 (Soundness).

(1) *If either* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$ *or* $\Gamma \vdash A \Longleftrightarrow B$ *then* $\Gamma \vdash A = B$.
(2) *If either* $\Gamma \vdash t \mathrel{\hat{\longrightarrow}} u : A$ *or* $\Gamma \vdash t \longrightarrow u : A$ *or* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ *or* $\Gamma \vdash t \Longleftrightarrow u : A$ *then* $\Gamma \vdash t = u : A$.

$\boxed{\Gamma \vdash n \longleftrightarrow m : A}$

$$\frac{\Gamma \vdash \mathsf{i}_x : A \qquad x \equiv y}{\Gamma \vdash \mathsf{i}_x \mathrel{\hat{\longleftrightarrow}} \mathsf{i}_y : A} \qquad \frac{\Gamma \vdash n \longleftrightarrow m : \Pi\, F\, G \qquad \Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : F}{\Gamma \vdash n\, t \mathrel{\hat{\longleftrightarrow}} m\, u : G[t]}$$

$$\frac{\begin{array}{c}\Gamma, \mathbb{N} \vdash F \mathrel{\hat{\Longleftrightarrow}} G \qquad \Gamma \vdash z \mathrel{\hat{\Longleftrightarrow}} z' : F[\mathsf{zero}] \\ \Gamma \vdash s \mathrel{\hat{\Longleftrightarrow}} s' : \Pi\, \mathbb{N}\, (F \to F[\uparrow \mathsf{id}, \mathsf{suc}\, i_0]) \qquad \Gamma \vdash n \longleftrightarrow m : \mathbb{N}\end{array}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, n \mathrel{\hat{\longleftrightarrow}} \mathsf{natrec}\, G\, z'\, s'\, m : F[n]}$$

$$\frac{\Gamma \vdash A \longrightarrow^* \overline{A} \qquad \Gamma \vdash n \mathrel{\hat{\longleftrightarrow}} m : A}{\Gamma \vdash n \longleftrightarrow m : \overline{A}}$$

$\boxed{\Gamma \vdash A \Longleftrightarrow B}$

$$\frac{\Gamma \vdash A \longrightarrow^* \overline{A} \qquad \Gamma \vdash B \longrightarrow^* \overline{B} \qquad \Gamma \vdash \overline{A} \Longleftrightarrow \overline{B}}{\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B} \qquad \frac{\Gamma \vdash N \longleftrightarrow M : \mathsf{U}}{\Gamma \vdash N \Longleftrightarrow M}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U} \Longleftrightarrow \mathsf{U}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} \Longleftrightarrow \mathbb{N}} \qquad \frac{\Gamma \vdash F \qquad \Gamma \vdash F \mathrel{\hat{\Longleftrightarrow}} H \qquad \Gamma, F \vdash G \mathrel{\hat{\Longleftrightarrow}} E}{\Gamma \vdash \Pi\, F\, G \Longleftrightarrow \Pi\, H\, E}$$

$\boxed{\Gamma \vdash t \Longleftrightarrow u : A}$

$$\frac{\Gamma \vdash A \longrightarrow^* \overline{A} \qquad \Gamma \vdash t \longrightarrow^* \overline{t} : \overline{A} \qquad \Gamma \vdash u \longrightarrow^* \overline{u} : \overline{A} \qquad \Gamma \vdash \overline{t} \Longleftrightarrow \overline{u} : \overline{A}}{\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : \mathbb{N}}{\Gamma \vdash n \Longleftrightarrow m : \mathbb{N}} \qquad \frac{\Gamma \vdash n : N \qquad \Gamma \vdash m : N \qquad \Gamma \vdash n \longleftrightarrow m : M}{\Gamma \vdash n \Longleftrightarrow m : N}$$

$$\frac{\Gamma \vdash \overline{A} : \mathsf{U} \qquad \Gamma \vdash \overline{B} : \mathsf{U} \qquad \Gamma \vdash \overline{A} \Longleftrightarrow \overline{B}}{\Gamma \vdash \overline{A} \Longleftrightarrow \overline{B} : \mathsf{U}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} \Longleftrightarrow \mathsf{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\, t \Longleftrightarrow \mathsf{suc}\, u : \mathbb{N}}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash \overline{f} : F \qquad \Gamma \vdash \overline{g} : F \qquad \Gamma \vdash \overline{f}[\uparrow \mathsf{id}]\, \mathsf{i}_0 \mathrel{\hat{\Longleftrightarrow}} \overline{g}[\uparrow \mathsf{id}]\, \mathsf{i}_0 : G}{\Gamma \vdash \overline{f} \Longleftrightarrow \overline{g} : \Pi\, F\, G}$$

Fig. 6. Algorithm for conversion of neutrals, types and terms.

Proof. By induction on the judgements and by syntactic validity (3.26), universe membership (3.34), neutral type equality (3.33), and reduction being subsumed by equality (2.2). □

Lemma 4.4 (Conversion).
Given $\vdash \Gamma = \Delta$ *and* $\Gamma \vdash A = B$ *and* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ *then* $\Delta \vdash t \mathrel{\hat{\Longleftrightarrow}} u : B$.

Proof. By induction on the judgements and by Π-injectivity (3.25), syntactic validity (3.26), weak head normalization (3.28), reduction being subsumed by equality (2.2) and context conversion (4.1 and 4.2). □

We can now prove decidability for the algorithm.

Theorem 4.5 (Decidability of algorithmic equality). *Given* $\vdash \Gamma = \Delta$*:*

(1) *If* $\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} t : A$ *and* $\Delta \vdash u \mathrel{\hat{\longleftrightarrow}} u : B$ *then it is decidable that there exists* C *such that* $\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} u : C$.
(2) *If* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} A$ *and* $\Delta \vdash B \mathrel{\hat{\Longleftrightarrow}} B$ *then* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$ *is decidable.*
(3) *If* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} t : A$ *and* $\Delta \vdash u \mathrel{\hat{\Longleftrightarrow}} u : A$ *then* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ *is decidable.*

Proof. By induction on the judgements and by Π-injectivity (3.25), syntactic validity (3.26), weak head normalization (3.28), strong equality (3.29), syntactic inequality (3.30), substitution (3.31), neutral type equality (3.33), determinism of reduction (2.5), soundness (4.3), context conversion for typing judgements (4.1) and conversion of algorithmic equality (4.4). □

Lemma 4.6 (Symmetry). *Given* $\vdash \Gamma = \Delta$*:*

(1) *If* $\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} u : A$ *then there exists* B *such that* $\Gamma \vdash A = B$ *and* $\Delta \vdash u \mathrel{\hat{\longleftrightarrow}} t : B$.
(2) *If* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$ *then* $\Delta \vdash B \mathrel{\hat{\Longleftrightarrow}} A$.
(3) *If* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ *then* $\Delta \vdash u \mathrel{\hat{\Longleftrightarrow}} t : A$.

Proof. By induction on the judgements and by Π-injectivity (3.25), syntactic validity (3.26), weak head normalization (3.28), strong equality (3.29), substitution (3.31), context conversion (4.1 and 4.2), soundness (4.3) and conversion of algorithmic equality (4.4). □

Lemma 4.7 (Transitivity). *Given* $\vdash \Gamma = \Delta$*:*

(1) *If* $\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} u : A$ *and* $\Delta \vdash u \mathrel{\hat{\longleftrightarrow}} v : A$ *then there exists* B *such that* $\Gamma \vdash A = B$ *and* $\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} v : B$.
(2) *If* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$ *and* $\Delta \vdash B \mathrel{\hat{\Longleftrightarrow}} C$ *then* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} C$.
(3) *If* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ *and* $\Delta \vdash u \mathrel{\hat{\Longleftrightarrow}} v : A$ *then* $\Delta \vdash t \mathrel{\hat{\Longleftrightarrow}} v : A$.

Proof. By induction on the judgements and by Π-injectivity (3.25), syntactic inequality (3.30), substitution (3.31), neutral type equality (3.33), reduction being subsumed by equality 2.2, determinism of reduction 2.5, context conversion (4.1 and 4.2) and soundness (4.3). □

Lemma 4.8 (Weakening). *For all algorithmic equality judgements* J*, given* $\rho : \Delta \leq \Gamma$ *and* $\Gamma \vdash J$ *then* $\Delta \vdash J[\rho]$.

Proof. By induction on the judgements and well-formed weakening (2.7). □

Lemma 4.9 (Whnf Lifting).

(1) *If* $\Gamma \vdash A \Longleftrightarrow B$ *then* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$.
(2) *If* $\Gamma \vdash t \Longleftrightarrow u : A$ *then* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$.

Proof. By syntactic validity (3.26) and soundness (4.3). □

Lemma 4.10 (Neutral lifting). *If* $\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} u : A$ *then* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$.

Proof. By well-formed objects being reducible (3.22), induction on the reducible type and then by syntactic validity (3.26), weak head normalization (3.28), reduction being subsumed by equality (2.2), determinism of reduction (2.5), reducible neutrals (3.10), and soundness (4.3). □

We can now construct our instance.

Instance 2 (Algorithmic equality instance). *The following instantiation of generic equality satisfies all the required properties for the logical relation:*

$$\Gamma \vdash A \cong B \text{ instantiated to } \Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$$
$$\Gamma \vdash t \cong u : A \text{ instantiated to } \Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$$
$$\Gamma \vdash t \sim u : A \text{ instantiated to the pair } \Gamma \vdash A = B \text{ and } \Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} u : B$$

Proof. We get the necessary properties from the definition of the algorithm and soundness (4.3), conversion (4.4), symmetry (4.6), transitivity (4.7), weakening (4.8), lifting (4.9) and neutral lifting (4.10) of algorithmic equality. □

With the instance of generic equality, we can now use the logical relation and the fundamental theorem to prove the completeness of algorithmic equality and finally decidability of conversion:

Theorem 4.11 (Completeness of algorithmic equality).

(1) *If* $\Gamma \vdash A = B$ *then* $\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$.
(2) *If* $\Gamma \vdash t = u : A$ *then* $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$.

Proof. By the fundamental theorem (3.21) with algorithmic equality (2) and escape (3.2). □

Theorem 4.12 (Decidability of conversion).

(1) *If* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *then* $\Gamma \vdash A = B$ *is decidable.*
(2) *If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash u : A$ *then* $\Gamma \vdash t = u : A$ *is decidable.*

Proof. By completeness (4.11), soundness (4.3) and decidability of algorithmic equality (4.5). □

5 CONCLUSION

We have fully formalized a substantial part of the meta-theory of a fragment of Martin-Löf type theory, using a limited set of features of Agda. Our formalization should be implementable in other type theories that support dependent types, universes, and induction-recursion.

However, the gap between expressive power of the formalized type theory (the object type theory) and the host type theory (the meta type theory) is fairly large: we require induction-recursion on the meta level to formalize just one inductive type and one universe on the object level. Yet the grand goal is *boot-strapping type theory*, meaning the implementation of type theory in a small extension of itself. (An extension in proof-theoretical strength is needed due to Gödel's incompleteness theorem.) A small extension would be, e.g., one more universe or one additional axiom. To advance towards the grand goal, we have to bring object and meta type theory closer together. Obviously, there are two directions: we can make the meta theory weaker, or the object theory stronger.

The main strength of the meta theory comes from induction-recursion [Dybjer 2000; Dybjer and Setzer 2001, 2003], which is a very powerful principle, proof-theoretically. Using iterated inductive-recursive definitions in the meta-theory with just one universe, we can model a countable hierarchy of universes of an object theory that lacks induction-recursion. Allowing more universes in the meta-theory, it is likely possible to encode our inductive-recursive logical relation using iterated inductive definitions only, placing the inductive definition of object-level universe n in meta-level universe n. Similar techniques have been used for the semantic modeling and auto-validation of impredicative type theories [Barras 2010; Werner 1994], using inaccessible cardinals which are analogous to universes. Eliminating induction-recursion would require reworking the central definition of the logical relation in our development, but could lead to a formalized boot-strapping of intensional predicative Martin-Löf Type Theory in itself (plus an extra universe and/or axiom).

Concerning the other direction, making the object theory stronger, this would amount to extending our logical relation to model induction-recursion. This is interesting future work, as it will enhance our understanding of induction-recursion from a syntactic perspective. Recent advances in understanding induction-recursion might aid this research [Ghani et al. 2015].

It remains interesting to investigate the relationship between the constructions in our proof and standard notions of models of type theory like Categories with Families (CwFs) by Dybjer [1995]. By contrast, Altenkirch and Kaposi [2016; 2017] directly define the syntax of type theory as an internal formulation of the initial CwF, i.e., quotienting terms by judgmental equality. In our case, however, we want to discuss reduction, which is only meaningful on terms that are not yet quotiented. A way to bridge this gap could be to formulate a weaker notion of CwFs, where some equations only have to hold up to some equivalence, and use it to restructure our proof, with the intent of providing a more abstract and generalizable presentation.

As for our contributions to proof methods, we have used proof-relevant induction-recursion to define a parameterized logical relation, which allows us to derive important properties like consistency, normalization, and Π-injectivity. We have managed to show that the *a priori* proof-relevant definitions are proof-irrelevant *a posteriori*, in the sense that the structure of the inductive derivations we recurse on does not matter in the end, only the judgement it derives (Lemma 3.5). Informally, this aspect is often glossed over; proof irrelevance is often assumed without officially being present in the meta-theory.

We have also parameterized this logical relation such that we only need to prove the fundamental theorem once, even though we have two slightly different instances. For the formalization, this means that we have saved a lot of work, as the parameterization does not add much code considering the size of the logical relation and the proof of the fundamental theorem, which add up to roughly 5000 lines of Agda code.

Our formalization comprises roughly 10.000 lines of Agda code (totaling 500.000 characters). It took 10 man-months of development work and type-checks on a contemporary laptop (16 GB memory, CPU speed 3.3GHz) in around 90 seconds. From this code we could extract the core of a type checker for fully-elaborated terms of a small fragment of the Agda language. It would neither be efficient nor could it do any type reconstruction, however, it could serve as a conceptual double checker of elaborated terms. In contrast, Agda itself consists roughly of 100.000 lines of Haskell code and features printer, parser, elaborator, termination and positivity analysis, pattern matcher and coverage checker, and many other components that are needed for a practically usable implementation of type theory. Verifying all these components with the same rigor as applied to our formalization seems clearly out of reach for an academic institution. However, a verified double checker for the full type theory behind Agda remains as a pursuable, grand challenge.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers, whose comments greatly helped to improve the presentation of our research in this article. The idea of basing the Kripke logical relation on a typed version of weak head reduction came from our colleagues Thierry Coquand and Bassel Mannaa. We also thank Paolo Capriotti for stimulating conversations on the relationship between notions of models and normalization. The first author acknowledges support by the Swedish Research Council (Vetenskapsrådet) under Grant No. 621-2014-4864 *Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types*. Our research groups are part of the EU Cost Action CA15123 *The European research network on types for programming and verification (EUTypes)*.

REFERENCES

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2007. Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society Press, 3–12. https://doi.org/10.1109/LICS.2007.33

Andreas Abel, Thierry Coquand, and Bassel Mannaa. 2016. On Decidability of Conversion in Type Theory. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016, Novi Sad, Serbia, May 23-26, 2016, Book of Abstracts*, Silvia Ghilezan and Jelena Ivetic (Eds.). EasyChair.

Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* 26 (2016), 61. https://doi.org/10.1017/S0956796816000022 ICFP 2013 special issue.

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1:29 (2012), 1–36. https://doi.org/10.2168/LMCS-8(1:29)2012 TYPES'10 special issue.

AgdaTeam. 2017. The Agda Wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM Press, 18–29. https://doi.org/10.1145/2837614.2837638

Thorsten Altenkirch and Ambrus Kaposi. 2017. Normalisation by Evaluation for Type Theory, in Type Theory. *Logical Methods in Computer Science* 13(4:1) (2017), 1–26. https://doi.org/10.23638/LMCS-13(4:1)2017

Bruno Barras. 2010. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning* 3, 1 (2010), 29–48. https://doi.org/10.6092/issn.1972-5787/1695

Thierry Coquand. 1991. An Algorithm for Testing Conversion in Type Theory. In *Logical Frameworks*, Gérard Huet and Gordon Plotkin (Eds.). Cambridge University Press, 255–279. http://dl.acm.org/citation.cfm?id=120477.120486

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

N. G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Peter Dybjer. 1995. Internal Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science)*, Stefano Berardi and Mario Coppo (Eds.), Vol. 1158. Springer, 120–134. https://doi.org/10.1007/3-540-61780-9_66

Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *The Journal of Symbolic Logic* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

Peter Dybjer and Anton Setzer. 2001. Indexed Induction-Recursion. In *Proof Theory in Computer Science, International Seminar, PTCS 2001, Dagstuhl Castle, Germany, October 7-12, 2001, Proceedings (Lecture Notes in Computer Science)*, Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk (Eds.), Vol. 2183. Springer, 93–113. https://doi.org/10.1007/3-540-45504-3_7

Peter Dybjer and Anton Setzer. 2003. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic* 124, 1-3 (2003), 1–47. https://doi.org/10.1016/S0168-0072(02)00096-9

Harvey Friedman. 1975. Equality between functionals. In *Logic Colloquium (Lecture Notes in Mathematics)*, R. Parikh (Ed.), Vol. 453. Springer, 22–37. https://doi.org/10.1007/BFb0064870

Herman Geuvers. 1994. A short and flexible proof of Strong Normalization for the Calculus of Constructions. In *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers (Lecture Notes in Computer Science)*, Peter Dybjer, Bengt Nordström, and Jan M. Smith (Eds.), Vol. 996. Springer, 14–38. https://doi.org/10.1007/3-540-60579-7_2

Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg. 2015. Positive Inductive-Recursive Definitions. 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:13)2015

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* Thèse de Doctorat d'État. Université de Paris VII.

Healfdene Goguen. 1994. *A Typed Operational Semantics for Type Theory.* Ph.D. Dissertation. University of Edinburgh. Available as LFCS Report ECS-LFCS-94-304.

Healfdene Goguen. 2000. A Kripke-Style Model for the Admissibility of Structural Rules. In *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers (Lecture Notes in Computer Science)*, Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack (Eds.), Vol. 2277. Springer, 112–124. https://doi.org/10.1007/3-540-45842-5_8

Robert Harper and Frank Pfenning. 2005. On Equivalence and Canonical Forms in the LF Type Theory. *ACM Trans. Comput. Logic* 6, 1 (Jan. 2005), 61–101. https://doi.org/10.1145/1042038.1042041

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*

Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 410–423. https://doi.org/10.1145/237721.240882

INRIA. 2017. *The Coq Proof Assistant Reference Manual* (version 8.7 ed.). INRIA. http://coq.inria.fr/

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson (Eds.). North-Holland, 73–118.

Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society Press, 233–242. https://doi.org/10.1109/LICS.2013.29

Carsten Schürmann and Jeffrey Sarnat. 2008. Structural Logical Relations. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, Frank Pfenning (Ed.). IEEE Computer Society Press, 69–80. https://doi.org/10.1109/LICS.2008.44

Thomas Streicher. 1993. Investigations into Intensional Type Theory. Habilitation thesis, Ludwig-Maximilians-University Munich.

Benjamin Werner. 1992. A Normalization Proof for an Impredicative Type System with Large Eliminations over Integers. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, Bengt Nordström, Kent Petersson, and Gordon Plotkin (Eds.). 341–357. http://www.cs.chalmers.se/Cs/Research/Logic/Types/proc92.ps

Benjamin Werner. 1994. *Une Théorie des Constructiones Inductives*. Ph.D. Dissertation. Universite Paris 7.