Codes for Quotient Inductive Inductive Types[⋆]

Ambrus Kaposi and András Kovács

Eötvös Loránd University, Budapest, Hungary
{akaposi|kovacsandras}@inf.elte.hu

Abstract. Quotient inductive inductive types (QIITs) are generalisations of inductive types in type theory. QIITs may consist of multiple sorts which can be indexed over each other. They also support equality constructors. In this paper we present a type of codes for QIITs along with interpretation functions which specify constructors, eliminators and computation rules for the encoded types. This is analogous to containers which act as codes for W-types. We present an internal syntax of a type theory with a universe and restricted function types. A code for a QIIT is a context in this type theory. The internal syntax is formalised as a QIIT itself. We consider a metatheory with unique identity proofs, hence we interpret codes as quotient instead of general higher inductive types. Some of the contents of this paper were formalised in the proof assistant Agda. Showing that the given QIITs exist is left as future work.

1 Introduction

Many dependent type theories support some form of inductive types. An inductive type is given by its constructors, along with an elimination principle which expresses that all inhabitants are constructed using finitely many applications of the constructors.

In this paper we present a specification for a large class of inductive types which encompasses indexed families, inductive inductive types and set-truncated higher inductive types, with possibly infinitary constructors. These types are called quotient inductive inductive types (QIITs).

Our method is based on the idea that the constructors of an inductive type can be specified as a context in a type theory. For example, natural numbers can be represented by a context of three entries:

$$\mathsf{Nat} : \mathsf{U}, \;\; \mathsf{zero} : \mathsf{Nat}, \;\; \mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}.$$

We define a syntax for a small type theory where each context encodes a QIIT. Additionally, this syntax is defined as a QIIT itself internally to a type-theoretic metatheory. Constructors and eliminators for specified types are given by translations from the internal syntax to the metatheory. Justifying the existence of the thus-specified inductive types is left to further work.

⋆ This work was supported by USAF grant FA9550-16-1-0029, Hungarian State Grant EFOP-3.6.3 and COST Action EUTypes CA15123.

We formalised the contents of this paper in the proof assistant Agda [24]. All of our constructions have been formalised using a shallow embedding. A fully precise formalisation using a deep embedding is underway. The formalisations are available at the first author's website.

Our contributions are:

- We give a general description of internal coding schemes for inductive types in section 3.
- We present a type theory where contexts specify QIITs, and provide several examples of encoded types in section 4.
- We derive elimination principles and computations rules for encoded types in sections 5–8.
- We extend the above construction with codes for infinitary constructors in section 9.
- We discuss how to extend our method to higher inductive inductive types in section 10.

The metatheory is described in section 2, and we discuss related work and conclude in section 11.

2 Metatheory and Notation

The metatheory is a type theory with a strict identity type (supporting uniqueness of identity proofs, or equivalently Streicher's axiom K [23]). To distinguish notation from the object theory, we generally denote metatheoretic colon by $\in$. There is a hierarchy of cumulative Russel-style universes Set_i, dependent function space $(\alpha \in A) \to B$, Σ types denoted by $(\alpha \in A) \times B$ with left-associative $\times$, the one-element type $\top$ with constructor tt and the identity type $=$ with constructor refl and eliminator J. The notation is $\mathsf{J}_{A\,t\,P}\,pr\,u\,eq$ for $t \in A$, $P \in (\alpha \in A) \to t = \alpha \to \mathsf{Set}$, $pr \in P\,t\,\mathsf{refl}$ and $eq \in t = u$. We write $\mathsf{tr}_P\,e\,u \in P\,\alpha'$ for transport of $u \in P\,\alpha$ along $e \in \alpha = \alpha'$. $A \to B$ stands for $(\alpha \in A) \to B$ if B does not mention α. We use $(\alpha \in A)(\beta \in B) \to C$ as a shorthand for iterated function spaces. Definitional equality in the metatheory is denoted by $\equiv$.

We use $\mathsf{ap}\,f\,e \in f\,u = f\,u'$ for $f \in A \to B$ and $e \in u = u'$, and $\mathsf{apd}\,f\,e \in \mathsf{tr}_P\,e\,(f\,u) = f\,u'$ when $f \in (\alpha \in A) \to B$. tr, ap and apd are defined using J in the standard way. We also use short versions for tr and J: $\mathsf{tr}\,e\,u$ with the first parameter omitted, and $\mathsf{J}\,pr\,eq$ with all parameters except the fourth and last omitted.

We require function extensionality, i. e. that the types $(\alpha \in A) \to f\,\alpha = g\,\alpha$ and $f = g$ are equivalent. The left-to-right direction is denoted funext.

The metatheory also supports at least one QIIT, the type of the syntax of codes described in section 4.

3 Coding Schemes for Inductive Types

Inductive types can be specified using external syntactic schemes or internal codes.

In the former case the type theory is extended with derivation rules specifying inductive types. External schemes for inductive families are given in [10, 20], for inductive recursive types in [11] and for a subset of higher inductive types in [5].

In the latter case there is an internal type of codes such that each code represents a valid inductive type, and actual types are produced from codes by decoding functions. Codes for simple inductive types such as natural numbers, lists or binary trees can be given by containers which are decoded to W-types [1]. Indexed families can be specified as indexed containers [18], and there are also codes for inductive recursive types [12], inductive inductive types [19] and another subset of higher inductive types [22].

We notice that internal coding schemes can be split into the components given in table 1. After explaining these components through the example of W-types [1], we present our coding scheme for QIITs by considering the same components in sections 4–8, respectively.

Type of codes	$\mathsf{Code} \in \mathsf{Set}$
Constructors	$-^{\mathsf{C}} \ \in \mathsf{Code} \to \mathsf{Set}$
Induction methods	$-^{\mathsf{M}} \ \in (\Gamma \in \mathsf{Code}) \to \Gamma^{\mathsf{C}} \to \mathsf{Set}$
Elimination principles	$-^{\mathsf{E}} \ \in (\Gamma \in \mathsf{Code})(\gamma \in \Gamma^{\mathsf{C}}) \to \Gamma^{\mathsf{M}}\,\gamma \to \mathsf{Set}$
Existence	$\mathsf{con} \ \in (\Gamma \in \mathsf{Code}) \to \Gamma^{\mathsf{C}}$
	$\mathsf{elim} \in (\Gamma \in \mathsf{Code})(m \in \Gamma^{\mathsf{M}}\,(\mathsf{con}\,\Gamma)) \to \Gamma^{\mathsf{E}}\,(\mathsf{con}\,\Gamma)\,m$

Table 1. Components of internal coding schemes for inductive types.

Each element of Code specifies an inductive type. In the case of W-types, Code is the type of containers: $\mathsf{Code} :\equiv (S \in \mathsf{Set}) \times (S \to \mathsf{Set})$. The two components are usually called "shapes" and "positions".

Given a code, the decoding function $-^{\mathsf{C}}$ provides the types of the type and value constructors, most conveniently as an iterated Σ type or a record of components. For W-types it is given as

$$(S, P)^{\mathsf{C}} :\equiv (W \in \mathsf{Set}) \times ((s \in S) \to (P\,s \to W) \to W).$$

The first component (W) is the type constructor and the second component is the value constructor (usually called *sup*).

The function $-^{\mathsf{M}}$ provides the types of the motives and methods of the eliminator. It depends on a choice of constructors. It returns a motive for each type constructor and a method for each value constructor. For the running example:

$$\begin{aligned}(S, P)^{\mathsf{M}}\,(W, sup) :\equiv{}& (W^M \in W \to \mathsf{Set})\\ &\times \big((s \in S)(f \in P\,s \to W) \to ((p \in P\,s) \to W^M\,(f\,p)) \to W^M\,(sup\,s\,f)\big)\end{aligned}$$

$-^{\mathsf{E}}$ provides the types of the elimination principles (one for each type constructor) and the computation rules as identity types. In the case of W-types:

$$(S,P)^{\mathsf{E}}\,(W, sup)\,(W^M, sup^M) :\equiv (W^E \in (w \in W) \to W^M\, w)$$
$$\times \big((s \in S)(f \in P\,s \to W) \to W^E\,(sup\,s\,f) = sup^M\,s\,f\,(\lambda p.W^E\,(f\,p))\big).$$

The first component W^E is the eliminator. Here it only has one input (the target) since the motives and methods are already given as arguments of $-^{\mathsf{E}}$. This would make especial sense for inductive inductive types which may have multiple eliminators which may share motives and methods. The second component returned by $-^{\mathsf{E}}$ is the computation rule. It explains the action of the eliminator on the constructor *sup*: the result is given by the application of the corresponding method sup^M and the recursive application of the eliminator.

There still remains the question whether the encoded types actually exist in a given type theory. If con and elim are inhabited, then they do exist. For example, in Agda or Coq the existence of W-types can be established with native inductive definitions. Alternatively, one could use external rules to postulate the existence of con and elim as a way of introducing new types.

4 Codes for QIITs

Codes for QIITs are contexts in the syntax of a small type theory which we call the *theory of codes*.

The theory of codes is given as a QIIT itself following [3]. It is intrinsically typed, i. e. there are no precontexts, pretypes or preterms, and we only consider well-formed constructions. The syntax has explicit substitutions. Conversion rules are given by equality constructors. There is an elimination principle which allows defining functions by induction on the syntax. Elimination ensures that all functions from the syntax respect typing and conversion rules. We will use this elimination principle to define the operations $-^{\mathsf{C}}$, $-^{\mathsf{M}}$, $-^{\mathsf{E}}$ in sections 5, 6, 7, respectively.

We believe that it is worth to use QIITs for internalising the theory of codes, since this allows us to do precise machine-checked formalisation of the development which would be prohibitively difficult otherwise.

For the current exposition, we eschew formal QIIT syntax in favor of a more familiar type-theoretic notation with variable names, implicit substitutions and implicit weakening. In particular, when we write horizontal lines for derivation rules, we mean metatheoretic function types. We also omit the rules for substitutions as they are standard.

The theory of codes has the following judgements.

$$\vdash \Gamma \qquad\qquad \Gamma \vdash A \qquad\qquad \Gamma \vdash t : A$$

Note that in QIIT notation these are the type constructors of an inductive inductive family, respectively:

$$\mathsf{Con} \in \mathsf{Set} \qquad\qquad \mathsf{Ty} \in \mathsf{Con} \to \mathsf{Set} \qquad\qquad \mathsf{Tm} \in (\Gamma \in \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$$

In our coding scheme, codes are contexts, hence $\mathsf{Code} :\equiv \mathsf{Con}$. Definitional equality in the theory of codes is denoted $=$ (the metatheoretic identity type).

The derivation rules for the theory of codes are given in figure 1. We explain the rules in order.

The rules for context formation and variables are standard. We assume fresh names everywhere to avoid name capture. Note that weakening is implicit.

There is a universe U, with decoding written as an underline (usually El in the literature). The purpose of U is to contain types which may serve as constructor arguments while preserving strict positivity.

Inductive function types have domains which may refer to constructors of the QIIT being defined, i. e. entries in the Γ context. The domain types must be in U, and since U is not closed under functions, function types cannot appear in inductive arguments, which ensures strict positivity. However, this also forces finitary constructors — infinitary constructors will be considered later in section 9. When the codomain does not depend on the domain, $a \to B$ can be written instead of $(x : a) \to B$.

Using this part of the syntax, we can already specify the code for natural numbers. It is given by the following context.

$$\cdot,\;\; nat : \mathsf{U},\;\; zero : \underline{nat},\;\; suc : nat \to \underline{nat}$$

We can also encode inductive inductive definitions such as the following fragment of an intrinsic syntax of a type theory.

$$\begin{aligned}&\cdot,\;\; Con : \mathsf{U},\;\; Ty : Con \to \mathsf{U},\;\; \bullet : \underline{Con},\;\; \rhd : (\Gamma : Con) \to Ty\,\Gamma \to \underline{Con},\\ &U : (\Gamma : Con) \to \underline{\mathsf{Ty}\,\Gamma},\;\; \Sigma : (\Gamma : Con)(A : Ty\,\Gamma)(B : Ty\,(\Gamma \rhd A)) \to \underline{Ty\,\Gamma}\end{aligned}$$

Using the full theory of codes, it is also possible to extend this to a code for the full syntax described in [3].

Note that we include λ in the syntax. This choice is not necessary for the purpose of encoding QIITs, since we are not aware of any example for which λ would be essential. Thus, λ and the corresponding conversion rules could be plausibly left out, but we include them anyway in order to make the theory of codes more complete as a stand-alone type theory. Also, it is perhaps a useful sanity check to define interpretations of λ-s as well.

There is another function space which we call *non-inductive*. We distinguish it from the inductive one by using $\in$ instead of colon in the domain specification. Given any metatheoretic type T in Set_0, we have non-inductive functions from that type. Their purpose is to include types from the metatheory which are external to the inductive type being defined. Note that in the type formation rule, $(\alpha \in T) \to \Gamma \vdash B_\alpha$ is a function in the metatheory which computes a syntactic type code for each α input. In this expression and in $(\alpha \in T) \to \Gamma \vdash t_\alpha : B_\alpha$ as well, Γ does not depend on α.

Now we can specify length-indexed vectors as follows. We need natural numbers $\mathbb{N}$ with 0 and $+1$ in the metatheory, as well as an $A \in \mathsf{Set}_0$ for the elements.

$$\cdot,\;\; vec : \mathbb{N} \to \mathsf{U},\;\; nil : \underline{vec\;0},\;\; cons : (n \in \mathbb{N})(a \in A) \to vec\;n \to \underline{vec\;(n+1)}$$

Contexts and variables:

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash x : A \qquad \Gamma \vdash B}{\Gamma, y : B \vdash x : A}$$

Universe:

$$\frac{}{\Gamma \vdash \mathsf{U}} \qquad \frac{\Gamma \vdash a : \mathsf{U}}{\Gamma \vdash \underline{a}}$$

Inductive functions:

$$\frac{\Gamma \vdash a : \mathsf{U} \qquad \Gamma, x : \underline{a} \vdash B}{\Gamma \vdash (x : a) \to B} \qquad \frac{\Gamma \vdash t : (x : a) \to B \qquad \Gamma \vdash u : \underline{a}}{\Gamma \vdash t\,u : B[x \mapsto u]}$$

$$\frac{\Gamma, x : \underline{a} \vdash t : B}{\Gamma \vdash \lambda x.t : (x : a) \to B} \qquad (\lambda x.t)\,u = t[x \mapsto u] \qquad \lambda x.(t\,x) = t$$

Non-inductive functions:

$$\frac{T \in \mathsf{Set}_0 \qquad (\alpha \in T) \to \Gamma \vdash B_\alpha}{\Gamma \vdash (\alpha \in T) \to B_\alpha} \qquad \frac{\Gamma \vdash t : (\alpha \in T) \to B_\alpha \qquad \alpha' \in T}{\Gamma \vdash t\,\alpha' : B_{\alpha'}}$$

$$\frac{(\alpha \in T) \to \Gamma \vdash t_\alpha : B_\alpha}{\Gamma \vdash \lambda\alpha.t_\alpha : (\alpha \in T) \to B_\alpha} \qquad (\lambda\alpha.t_\alpha)\,\alpha' = t_{\alpha'} \qquad \lambda\alpha.(t\,\alpha) = t$$

Identity:

$$\frac{\Gamma \vdash a : \mathsf{U} \qquad \Gamma \vdash t : \underline{a} \qquad \Gamma \vdash u : \underline{a}}{\Gamma \vdash \mathsf{Id}_a\,t\,u : \mathsf{U}} \qquad \frac{\Gamma \vdash t : \underline{a}}{\Gamma \vdash \mathsf{refl} : \underline{\mathsf{Id}_a\,t\,t}}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : \underline{a} \\ \Gamma, x : \underline{a}, z : \underline{\mathsf{Id}_a\,t\,x} \vdash P \\ \Gamma \vdash pr : P[x \mapsto t, z \mapsto \mathsf{refl}] \\ \Gamma \vdash u : \underline{a} \\ \Gamma \vdash eq : \underline{\mathsf{Id}_a\,t\,u}\end{array}}{\Gamma \vdash \mathsf{J}_{a\,t\,(x.z.P)}\,pr\,u\,eq : P[x \mapsto u, z \mapsto eq]}$$

$$\mathsf{J}_{a\,t\,(x.z.P)}\,pr\,t\,\mathsf{refl} = pr$$

Fig. 1. The theory of codes.

For another example, propositional equality for an $A \in \mathsf{Set}_0$ is encoded as follows.

$$\cdot,\ \ Eq : (x \in A)(y \in A) \to \mathsf{U},\ \ refl : (x \in A) \to \underline{Eq\,x\,x}$$

Smallness of non-inductive argument types is assumed for simplicity. It is possible to generalize codes to arbitrary universe levels, but it is not essential to the current development. Currently, type constructors parametrized by metatheoretic types are not possible since their type is not small: for example, the type of *vec* cannot be $(A \in \mathsf{Set}) \to \mathbb{N} \to \mathsf{U}$. This is not a significant limitation, since as shown in the *vec* example, it is always possible to construct codes in the metatheory with parameters instantiated as needed. Having codes on any universe level would be mainly interesting for the fact that it would allow self-describing or "levitated" [8] codes as well, when combined with codes for infinitary constructors. This is because the QIIT of codes is large (since it refers to metatheoretic types) and infinitary as well (since it refers to metatheoretic functions which return derivations in the theory of codes).

Note that non-inductive functions preserve strict positivity, since there is no way to recursively refer to the code of the inductive type *being defined* in the metatheory. The situation is analogous to the case of W-types, where codes contain metatheoretic types and type families for shapes and positions respectively, but neither can recursively refer to the code itself.

Furthermore, U is closed under identity (or equality), with large elimination J, allowing codes for higher constructors and recursive identity types in constructors. We list the codes for several higher inductive types. The circle:

$$\cdot,\ \ s^1 : \mathsf{U},\ \ base : \underline{s^1},\ \ loop : \underline{\mathsf{Id}_{s^1}\,base\,base}$$

Propositional truncation for a metatheoretic $A \in \mathsf{Set}_0$:

$$\cdot,\ \ tr : \mathsf{U},\ \ emb : (a \in A) \to \underline{tr},\ \ eq : (x\,y : tr) \to \underline{\mathsf{Id}_{tr}\,x\,y}$$

Set truncation likewise:

$$\cdot,\ \ tr : \mathsf{U},\ \ emb : (a \in A) \to \underline{tr},\ \ eq : (x\,y : tr)(p\,q : \mathsf{Id}_{tr}\,x\,y) \to \underline{\mathsf{Id}_{\mathsf{Id}_{tr}\,x\,y}\,p\,q}$$

The J rule allows constructors to mention operations on paths as well. For instance, the definition of the torus depends on path composition, which can be defined using J. Using $\cdot$ for composition, the code for the torus is given as follows [21, p. 193]:

$$\cdot,\ \ t^2 : \mathsf{U},\ \ b : \underline{t^2},\ \ p : \underline{\mathsf{Id}_{t^2}\,b\,b},\ \ q : \underline{\mathsf{Id}_{t^2}\,b\,b},\ \ t : \underline{\mathsf{Id}_{\mathsf{Id}_{t^2}\,b\,b}\,(p \cdot q)\,(q \cdot p)}$$

In summary, a code for an inductive type is given by a context in the theory of codes, and each entry specifies a constructor. In general, the return type in a context entry is of three possible forms: it is either U, $\underline{a}$ for some neutral a, or $\underline{\mathsf{Id}_a\,t\,u}$. In sections 4–8 we present the $-^{\mathsf{C}}$, $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ operations, which respectively compute constructors, induction methods and elimination principles from codes. The following table summarizes what these operations compute in the mentioned three cases:

return type	$-^{\mathsf{C}}$	$\_^{\mathsf{M}}$	$\_^{\mathsf{E}}$
U	type constructor	motive	eliminator
$\underline{a}$	point constructor	method	computation rule
$\underline{\mathsf{Id}_a\, t\, u}$	path constructor	method expressing preservation of equality	computation rule

Note that there is no syntactic distinction between the three kinds of constructors above. Any number of them can be introduced in any order, and each constructor can refer to any previous one. A distinguishing feature of our approach is the utilisation of universes instead of structural rules to introduce new sorts and to ensure strict positivity.

Also, note the higher equality constructors in the set truncation and torus examples. In the current development, higher equalities become trivial when interpreted in the metatheory, since the metatheory has uniqueness of identity proofs. Nevertheless, for higher equalities our interpretation functions compute constructors and eliminators which conform to prior literature on higher inductive types. See section 10 for possible ways to extend the current development to proof-relevant higher equalities.

5 Constructors

The operation $-^{\mathsf{C}}$, given a context in the theory of codes, returns the types of constructors in the metatheory. For example, it interprets the code for natural numbers as follows:

$$(\cdot,\, n : \mathsf{U},\, z : \underline{n},\, s : n \to \underline{n})^{\mathsf{C}} \equiv \top \times (n \in \mathsf{Set}_0) \times (z \in n) \times (s \in n \to n)$$

$-^{\mathsf{C}}$ is defined by mutual recursion on contexts, types and terms in the theory of codes. It is specified as follows.

$$\frac{\vdash \Gamma}{\Gamma^{\mathsf{C}} \in \mathsf{Set}_1} \qquad \frac{\Gamma \vdash A}{A^{\mathsf{C}} \in \Gamma^{\mathsf{C}} \to \mathsf{Set}_1} \qquad \frac{\Gamma \vdash t : A}{t^{\mathsf{C}} \in (\gamma \in \Gamma^{\mathsf{C}}) \to A^{\mathsf{C}}\,\gamma}$$

The implementation corresponds to the interpretation into the standard model (set model) where every object-theoretic construction is mapped to its metatheoretic counterpart. Contexts are interpreted as left-nested iterated Σ types, i.e. $\cdot^{\mathsf{C}} :\equiv \top$ and $(\Gamma, x : A)^{\mathsf{C}} :\equiv (\gamma \in \Gamma^{\mathsf{C}}) \times A^{\mathsf{C}}\,\gamma$. In the above example for natural numbers, $\top \times (n \in \mathsf{Set}) \times (z \in n) \times (s \in n \to n)$ is meant to be a shorthand for the left-nested Σ type where previous components are referred to by projections instead of variables. The full definition of $-^{\mathsf{C}}$ is available in the appendix A.

All definitional equalities in codes are preserved by $-^{\mathsf{C}}$.

6 Induction Methods

Given a code for an inductive type and the constructors specified by the code, the operation $-^{\mathsf{M}}$ returns the motives and methods for the eliminator. In our setting, induction motives are just induction methods for type constructors, hence, we shall only talk about induction methods from now on. $-^{\mathsf{M}}$ works on the code for natural numbers as follows:

$$\begin{aligned}&(\cdot,\, nat : \mathsf{U},\, zero : \underline{nat},\, suc : nat \to \underline{nat})^{\mathsf{M}}\,(\mathsf{tt}, n, z, s)\\ \equiv\ &\top \times (n^{\mathsf{M}} \in n \to \mathsf{Set}_i) \times (z^{\mathsf{M}} \in n^{\mathsf{M}}\, z) \times \big(s^{\mathsf{M}} \in (\alpha \in n) \to n^{\mathsf{M}}\,\alpha \to n^{\mathsf{M}}\,(s\,\alpha)\big)\end{aligned}$$

$-^{\mathsf{M}}$ is a variant of the unary logical predicate translation of Bernardy et al. [6]. We fix a level i for the universe we would like to eliminate into. For each context $\mathit{\Gamma}$, $\mathit{\Gamma}^{\mathsf{M}}$ is a predicate over the standard interpretation $\mathit{\Gamma}^{\mathsf{C}}$. For a type $\mathit{\Gamma} \vdash A$, A^{M} is a predicate over the standard interpretation A^{C}, which also depends on the standard interpretation $\gamma \in \mathit{\Gamma}^{\mathsf{C}}$ and a witness of $\mathit{\Gamma}^{\mathsf{M}}\,\gamma$.

$$\frac{\vdash \mathit{\Gamma}}{\mathit{\Gamma}^{\mathsf{M}} \in \mathit{\Gamma}^{\mathsf{C}} \to \mathsf{Set}_{i+1}} \qquad \frac{\mathit{\Gamma} \vdash A}{A^{\mathsf{M}} \in (\gamma \in \mathit{\Gamma}^{\mathsf{C}}) \to \mathit{\Gamma}^{\mathsf{M}}\,\gamma \to A^{\mathsf{C}}\,\gamma \to \mathsf{Set}_{i+1}}$$

For a term t, t^{M} witnesses that the predicate corresponding to its type holds for t^{C}. This is often called a *fundamental theorem* in the literature on logical predicates.

$$\frac{\mathit{\Gamma} \vdash t : A}{t^{\mathsf{M}} \in (\gamma \in \mathit{\Gamma}^{\mathsf{C}})(\gamma^M \in \mathit{\Gamma}^{\mathsf{M}}\,\gamma) \to A^{\mathsf{M}}\,\gamma\,\gamma^M\,(t^{\mathsf{C}}\,\gamma)}$$

We introduce the following shorthand: $t\,\gamma\,\gamma^M$ is abbreviated as $t\,\gamma^2$ for some t expression. The implementation of $-^{\mathsf{M}}$ is given below.

$$\begin{aligned}
&\cdot^{\mathsf{M}}\,\gamma &&:\equiv \top\\
&(\mathit{\Gamma},\, x : A)^{\mathsf{M}}\,(\gamma,\, \alpha) &&:\equiv (\gamma^M \in \mathit{\Gamma}^{\mathsf{M}}\,\gamma) \times A^{\mathsf{M}}\,\gamma^2\,\alpha\\
&x^{\mathsf{M}}\,\gamma^2 &&:\equiv x^{\text{th}} \text{ component in } \gamma^M\\
&\mathsf{U}^{\mathsf{M}}\,\gamma^2\,T &&:\equiv T \to \mathsf{Set}_i\\
&(\underline{a})^{\mathsf{M}}\,\gamma^2\,\alpha &&:\equiv a^{\mathsf{M}}\,\gamma^2\,\alpha\\
&((x : a) \to B)^{\mathsf{M}}\,\gamma^2\,f &&:\equiv (\alpha \in a^{\mathsf{C}}\,\gamma)(\alpha^M \in a^{\mathsf{M}}\,\gamma^2\,\alpha)\\
& &&\quad\ \to B^{\mathsf{M}}\,(\gamma, \alpha)\,(\gamma^M, \alpha^M)\,(f\,\alpha)\\
&(t\,u)^{\mathsf{M}}\,\gamma^2 &&:\equiv (t^{\mathsf{M}}\,\gamma^2)\,(u^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{M}}\,\gamma^2)\\
&(\lambda x.t)^{\mathsf{M}}\,\gamma^2 &&:\equiv \lambda\alpha\,\alpha^M.t^{\mathsf{M}}\,(\gamma, \alpha)\,(\gamma^M, \alpha^M)\\
&((\alpha \in T) \to B_\alpha)^{\mathsf{M}}\,\gamma^2\,f &&:\equiv (\alpha \in T) \to (B_\alpha)^{\mathsf{M}}\,\gamma^2\,(f\,\alpha)\\
&(t\,\alpha)^{\mathsf{M}}\,\gamma^2 &&:\equiv t^{\mathsf{M}}\,\gamma^2\,\alpha\\
&(\lambda\alpha.t_\alpha)^{\mathsf{M}}\,\gamma^2 &&:\equiv \lambda\alpha.(t_\alpha)^{\mathsf{M}}\,\gamma^2\\
&(\mathsf{Id}_a\,t\,u)^{\mathsf{M}}\,\gamma^2 &&:\equiv \lambda e.\mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,e\,(t^{\mathsf{M}}\,\gamma^2) = u^{\mathsf{M}}\gamma^2
\end{aligned}$$

$$\begin{aligned}
&(\mathsf{refl}_t)^{\mathsf{M}}\,\gamma^2 && :\equiv \mathsf{refl}_{(t^{\mathsf{M}}\,\gamma^2)} \\
&(\mathsf{J}_{a\,t\,(x.z.P)}\,pr\,u\,eq)^{\mathsf{M}}\,\gamma^2 && :\equiv \mathsf{J}\,\big(\mathsf{J}\,(pr^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{C}}\,\gamma)\big)\,(eq^{\mathsf{M}}\,\gamma^2)
\end{aligned}$$

The predicate for a context is given by iterating $-^{\mathsf{M}}$ for its constituent types. For a variable, the corresponding witness is looked up from γ^{M}.

The predicate for the universe, given an element of $T \in \mathsf{U}^{\mathsf{C}}\,\gamma$ (with $\mathsf{U}^{\mathsf{C}}\,\gamma = \mathsf{Set}_0$) returns the predicate space over T. The predicate for a type $\underline{a}$ is given by the predicate for a.

The predicate for inductive function types expresses preservation of predicates (at the domain and codomain types). Witnesses of application and abstraction are given by recursive application of $-^{\mathsf{M}}$. The definitions for non-inductive functions are similar, except there is no predicate for the metatheoretic domain type, and thus no witnesses are required.

The predicate for the identity type $\mathsf{Id}_a\,t\,u$, for each $e \in (\mathsf{Id}_a\,t\,u)^{\mathsf{C}}\,\gamma$, i. e. $e \in t^{\mathsf{C}}\,\gamma = u^{\mathsf{C}}\,\gamma$, says t^{M} and u^{M} are equal. As these have different types, we have to transport over the original equality e. The witness for refl is reflexivity in the metatheory. The witness for J is given by double application of the metatheoretic J. The definition is sourced from [16]. Here, we use a shortened J notation; the full definition can be found in appendix C.

All definitional equalities are preserved by $-^{\mathsf{M}}$.

7 Elimination Principles

The operation $-^{\mathsf{E}}$ yields eliminators and computation rules. It works as follows on the code for natural numbers:

$$\begin{aligned}
&(\cdot, nat : \mathsf{U}, zero : \underline{nat}, suc : nat \to \underline{nat})^{\mathsf{E}}\,(\mathsf{tt}, n, z, s)\,(\mathsf{tt}, n^{M}, z^{M}, s^{M}) \\
&\equiv \top \times \big(n^{E} \in (\alpha \in n) \to n^{M}\,\alpha\big) \times (z^{E} \in n^{E}\,z = z^{M}) \\
&\quad\; \times \big(s^{E} \in (\alpha \in n) \to n^{E}\,(s\,\alpha) = n^{M}\,\alpha\,(n^{E}\,\alpha)\big)
\end{aligned}$$

Previously, the $-^{\mathsf{C}}$ and $-^{\mathsf{M}}$ operations both returned iterated Σ types of the same length as the input context, yielding a constructor and an induction method respectively for each entry. $-^{\mathsf{E}}$ provides the type of the eliminator for each type constructor and the computation rule (β-rule) for each path and point constructor. The computation rules are given using the metatheoretic identity type $=$.

$-^{\mathsf{E}}$ can be viewed as a generalised binary logical relation translation where the type of the second parameter in the relation may depend on the first parameter.

Contexts are interpreted as dependent binary relations between constructors and methods. The universe level i was previously chosen for the $-^{\mathsf{M}}$ operation.

$$\frac{\vdash \Gamma}{\Gamma^{\mathsf{E}} \in (\gamma \in \Gamma^{\mathsf{C}}) \to \Gamma^{\mathsf{M}}\,\gamma \to \mathsf{Set}_i}$$

Types are interpreted as dependent binary relations which additionally depend on (γ, γ^{M}, γ^{E}) interpretations of the context.

$$\frac{\Gamma \vdash A}{A^{\mathsf{E}} \in (\gamma \in \Gamma^{\mathsf{C}})(\gamma^{M} \in \Gamma^{\mathsf{M}}\,\gamma)(\gamma^{E} \in \Gamma^{\mathsf{E}}\,\gamma\,\gamma^{M})(\alpha \in A^{\mathsf{C}}\,\gamma) \to A^{\mathsf{M}}\,\gamma\,\gamma^{M}\,\alpha \to \mathsf{Set}_i}$$

For a term t, t^{E} witnesses that the relation corresponding to its type holds for t^{C} and t^{M}.

$$\frac{\Gamma \vdash t : A}{t^{\mathsf{E}} \in (\gamma \in \Gamma^{\mathsf{C}})(\gamma^M \in \Gamma^{\mathsf{M}}\,\gamma)(\gamma^E \in \Gamma^{\mathsf{E}}\,\gamma\,\gamma^M) \to A^{\mathsf{E}}\,\gamma\,\gamma^M\,\gamma^E\,(t^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{M}}\,\gamma\,\gamma^M)}$$

In addition to γ^2, we use $t\,\gamma^3$ to abbreviate $t\,\gamma\,\gamma^M\,\gamma^E$. The implementation is the following.

$$\begin{array}{ll}
\cdot^{\mathsf{E}}\;\gamma\,\gamma^M & :\equiv \top \\
(\Gamma,\,x : A)^{\mathsf{E}}\,(\gamma,\alpha)\,(\gamma^M,\alpha^M) & :\equiv (\gamma^E \in \Gamma^{\mathsf{E}}\gamma^2) \times A^{\mathsf{E}}\,\gamma^3\,\alpha\,\alpha^M \\
x^{\mathsf{E}}\,\gamma^3 & :\equiv x^{\text{th}} \text{ component in } \gamma^E \\
\mathsf{U}^{\mathsf{E}}\,\gamma^3\,T\,T^M & :\equiv (\alpha \in T) \to T^M\,\alpha \\
(\underline{a})^{\mathsf{E}}\,\gamma^3\,\alpha\,\alpha^M & :\equiv a^{\mathsf{E}}\,\gamma^3\,\alpha = \alpha^M \\
((x : a) \to B)^{\mathsf{E}}\,\gamma^3\,f\,f^M & :\equiv (\alpha \in a^{\mathsf{C}}\,\gamma) \to B^{\mathsf{E}}\,(\gamma,\alpha)\,(\gamma^M, a^{\mathsf{E}}\,\gamma^3\,\alpha) \\
 & \qquad\qquad (\gamma^E,\mathsf{refl})\,(f\,\alpha)\,\big(f^M\,\alpha\,(a^{\mathsf{E}}\,\gamma^3\,\alpha)\big) \\
(t\,u)^{\mathsf{E}}\,\gamma^3 & :\equiv \mathsf{J}\,(t^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma))\,(u^{\mathsf{E}}\,\gamma^3) \\
(\lambda x.t)^{\mathsf{E}}\,\gamma^3 & :\equiv \lambda\alpha.t^{\mathsf{E}}\,(\gamma,\alpha)\,(\gamma^M, a^{\mathsf{E}}\,\gamma^3\,\alpha)\,(\gamma^E,\mathsf{refl}) \\
((\alpha \in T) \to B_\alpha)^{\mathsf{E}}\,\gamma^3\,f\,f^M & :\equiv (\alpha \in T) \to (B_\alpha)^{\mathsf{E}}\,\gamma^3\,(f\,\alpha)\,(f^M\,\alpha) \\
(t\,\alpha)^{\mathsf{E}}\,\gamma^3 & :\equiv t^{\mathsf{E}}\,\gamma^3\,\alpha \\
(\lambda\alpha.t_\alpha)^{\mathsf{E}}\,\gamma^3 & :\equiv \lambda\alpha.(t_\alpha)^{\mathsf{E}}\,\gamma^3 \\
(\mathsf{Id}_a\,t\,u)^{\mathsf{E}}\,\gamma^3 & :\equiv \lambda e.\mathsf{tr}\,(t^{\mathsf{E}}\,\gamma^3)\Big(\mathsf{tr}\,(u^{\mathsf{E}}\,\gamma^3)\big(\mathsf{apd}\,(a^{\mathsf{E}}\,\gamma^3)\,e\big)\Big) \\
(\mathsf{refl}_t)^{\mathsf{E}}\,\gamma^3 & :\equiv \mathsf{J}\,\mathsf{refl}\,(t^{\mathsf{E}}\,\gamma^3) \\
(\mathsf{J}_{a\,t\,(x.z.P)}\,pr\,u\,eq)^{\mathsf{E}}\,\gamma^3 & :\equiv
\end{array}$$

$$\mathsf{J}\,\Bigg(\mathsf{J}\,\Bigg(\mathsf{J}\,\big(\mathsf{J}\,(\lambda\,P^M P^E pr^M pr^E \to pr^E)\,(t^{\mathsf{E}}\,\gamma^3)$$

$$(\mathsf{uncurry}\,P^{\mathsf{M}}\,\gamma^2)\,(\mathsf{uncurry}\,P^{\mathsf{E}}\,\gamma^3)\,(pr^{\mathsf{M}}\,\gamma^2)\,(pr^{\mathsf{E}}\,\gamma^3)\big)\,eq^{\mathsf{C}}\,\gamma\Bigg)\,u^{\mathsf{E}}\,\gamma^3\Bigg)\,(eq^{\mathsf{E}}\,\gamma^3)$$

The U^{E} and $(\underline{a})^{\mathsf{E}}$ definitions are the key points of the $-^{\mathsf{E}}$ operation. The former computes the types of eliminators for type constructors, while the latter computes the types of β-rules for point and path constructors. The definitions for the other $-^{\mathsf{E}}$ cases are largely determined by these.

The U^{E} rule yields the expected types of eliminators for non-indexed type constructors. The types of eliminators for *indexed* type constructors follow from the $((x : a) \to B)^{\mathsf{E}}$ and $((\alpha \in T) \to B_\alpha)^{\mathsf{E}}$ cases: if a function type eventually returns in U, then the U^{E} rule will apply at the end, yielding the appropriately indexed type for the eliminator.

For point and path constructors, we get the type of computation rules from $(\underline{a})^{\mathsf{E}}$, which expresses that applying the eliminator a^{E} on α equals the corresponding α^M method. Again, the operations for function types provide the computation rules for point and path constructors with multiple arguments.

In the definition of $((x : a) \to B)^{\mathsf{E}}$, the resulting function type only takes a single $(\alpha \in a^{\mathsf{C}}\,\Gamma)$ as input, and it *does not* abstract over $-^{\mathsf{M}}$ witnesses, unlike as we have seen for $((x : a) \to B)^{\mathsf{M}}$. This is because we can freely generate witnesses using the a^{E} *eliminator*. The reason we have a^{E} as an eliminator is because the domains of inductive functions are restricted to U. If we had a general A type instead of an $a : \mathsf{U}$ code, A^{E} would only yield a *relation* instead. In our definition, we use a^{E} to provide a witness for $a^{\mathsf{M}}\,\gamma\,\alpha$ in $(\gamma^M, a^{\mathsf{E}}\,\gamma^3\,\alpha)$. We also need to extend γ^{E} with a witness for a β-rule expressing that using the a^{E} eliminator on α equals the induction method for α. However, we have just specified the induction method as $a^{\mathsf{E}}\,\gamma^3\,\alpha$, so the witness for the β-rule can be given as refl.

In $(t\,u)^{\mathsf{E}}$, the definition requires a J usage. Clearly, we need to apply t^{E} to some $(\alpha \in a^{\mathsf{C}}\,\gamma)$, so we may attempt to give $t^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma)$ as definition. However, it follows from $((x : a) \to B)^{\mathsf{E}}$ that this has type

$$B^{\mathsf{E}}\,(\gamma, u^{\mathsf{C}}\,\gamma)\,(\gamma^{\mathsf{M}}, a^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma)))\,(\gamma^E, \mathsf{refl})\,(t^{\mathsf{C}}\,\gamma\,(u^{\mathsf{C}}\,\gamma))\,(t^{\mathsf{M}}\,\gamma^2\,(u^{\mathsf{C}}\,\gamma)\,(a^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma)))),$$

but the required type is

$$B^{\mathsf{E}}\,(\gamma, u^{\mathsf{C}}\,\gamma)\,(\gamma^{\mathsf{M}}, u^{\mathsf{M}}\,\gamma^2)\,(\gamma^E, u^{\mathsf{E}}\,\gamma^3)\,(t^{\mathsf{C}}\,\gamma\,(u^{\mathsf{C}}\,\gamma))\,(t^{\mathsf{M}}\,\gamma^2\,(u^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{M}}\,\gamma^2)).$$

We also know that $u^{\mathsf{E}}\,\gamma^3 \in a^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma) = u^{\mathsf{M}}\,\gamma^2$. Hence we need to use path induction on $u^{\mathsf{E}}\,\gamma^3$ to get the desired type; see appendix D for the full definition. From another perspective, J is needed in the definition because u^{E} yields a witness for a β-rule, i. e. an equality proof, and the only way to make use of it is through J. Likewise, t^{E} yields a function, and the only way to use it in the definition is to apply it.

$(\mathsf{Id}_a\,t\,u)^{\mathsf{E}}\,\gamma^3$ needs to provide a witness for $(\mathsf{Id}_a\,t\,u)^{\mathsf{M}}\,\gamma^2\,e$ for each e with type $t^{\mathsf{C}}\,\gamma = u^{\mathsf{C}}\,\gamma$. Unfolding the goal type, we get $\mathsf{tr}\,e\,(t^{\mathsf{M}}\,\gamma^2) = u^{\mathsf{M}}\gamma^2$. We use $\mathsf{apd}\,(a^{\mathsf{E}}\,\gamma^3)\,e$ to get $a^{\mathsf{M}}\,\gamma^2$ witnesses on both sides of the equation, then we transport twice with β-equalities given by $u^{\mathsf{E}}\,\gamma^3$ and $t^{\mathsf{E}}\,\gamma^3$ to rewrite the equation sides to the needed form.

For $(\mathsf{refl}_t)^{\mathsf{E}}$, we need to witness the corresponding β-rule which requires that $(\mathsf{Id}_a\,t\,t)^{\mathsf{E}}\,\gamma^3\,\mathsf{refl}$ equals $\mathsf{refl}^{\mathsf{M}}\,\gamma^2$. The right hand side is just refl, while the left hand side expands to a doubly transported refl, with both transports going over $t^{\mathsf{E}}\,\gamma^3$. Hence a single J over $t^{\mathsf{E}}\,\gamma^3$ lets us to prove the equality with refl.

We write the definition for $(\mathsf{J}_{T\,t\,P}\,pr\,u\,eq)^{\mathsf{E}}$ here with the short J notation. Also, we write $\mathsf{uncurry}\,P^{\mathsf{M}}$ for $\lambda\,\gamma\,\gamma^M\,x\,x^M\,z\,z^M.P^{\mathsf{M}}\,(\gamma, x, z)\,(\gamma^M, x^M, z^M)$ and analogously for $\mathsf{uncurry}\,P^{\mathsf{E}}$. The full definition is rather complex and difficult to handle outside of a proof assistant; it is available in the shallow Agda formalisation. In short, in order to provide the definition, we need to use J on all available equality proofs, namely $eq^{\mathsf{E}}\,\gamma^3$, $u^{\mathsf{E}}\,\gamma^3$, $eq^{\mathsf{C}}\,\gamma$ and $t^{\mathsf{E}}\,\gamma^3$, in this order. In the innermost J expression, we additionally need to generalise over P^M, P^E, pr^M and pr^E in order to rewrite their types along $t^{\mathsf{E}}\,\gamma^3$.

Again, definitional equalities are preserved by $-^{\mathsf{E}}$, although in this case preservation proofs depend on β-equalities given by $-^{\mathsf{E}}$ as well.

8 Existence of QIITs

Rephrasing the requirements for existence in table 1, we say that the metatheory supports QIITs if the following rules are admissible.

$$\frac{\vdash \Gamma}{\mathsf{con}_\Gamma \in \Gamma^{\mathsf{C}}} \qquad \frac{\vdash \Gamma \qquad m \in \Gamma^{\mathsf{M}}\,\mathsf{con}_\Gamma}{\mathsf{elim}_\Gamma\, m \in \Gamma^{\mathsf{E}}\,\mathsf{con}_\Gamma\, m}$$

9 Infinitary Constructors

In this section we add codes for infinitary constructors, i. e. constructors which have function arguments with strictly positive recursive occurrences of type constructors. This can be accomplished by adding *small non-inductive functions.*

$$\frac{T \in \mathsf{Set}_0 \qquad (\alpha \in T) \to \Gamma \vdash b_\alpha : \mathsf{U}}{\Gamma \vdash (\alpha \in T) \to b_\alpha : \mathsf{U}} \qquad \frac{\Gamma \vdash t : \underline{(\alpha \in T) \to b_\alpha} \qquad \alpha' \in T}{\Gamma \vdash t\,\alpha' : \underline{b_{\alpha'}}}$$

$$\frac{(\alpha \in T) \to \Gamma \vdash t_\alpha : \underline{b_\alpha}}{\Gamma \vdash \lambda\alpha.t_\alpha : \underline{(\alpha \in T) \to b_\alpha}} \qquad (\lambda\alpha.t_\alpha)\,\alpha' = t_{\alpha'} \qquad \lambda\alpha.(t\,\alpha) = t$$

Small non-inductive functions differ from non-inductive functions by returning in some $b_\alpha : \mathsf{U}$ and by being in U themselves. These two features ensure strict positivity while allowing functions with inductive return types in constructor arguments. We overload the notation for abstraction and application since it can be usually inferred from context whether a non-inductive function is small.

Now, assuming $S \in \mathsf{Set}_0$ and $P \in S \to \mathsf{Set}_0$, the code of the corresponding W-type is as follows:

$$W_{SP} :\equiv (\cdot,\ \ w : \mathsf{U},\ \ sup : (s \in S) \to ((p \in P) \to w) \to \underline{w})$$

Here, $(p \in P) \to w$ is a small non-inductive function in U, hence it can be used as domain type in the inductive $((p \in P) \to w) \to \underline{w}$ function.

The $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ definitions are the following ($-^{\mathsf{C}}$ is given in appendix A).

$$\begin{aligned}
((\alpha \in T) \to b_\alpha)^{\mathsf{M}}\,\gamma^2 &:\equiv \lambda f.\,(\alpha \in T) \to (b_\alpha)^{\mathsf{M}}\,\gamma^2\,(f\,\alpha)\\
(t\,\alpha)^{\mathsf{M}}\,\gamma^2 &:\equiv t^{\mathsf{M}}\,\gamma^2\,\alpha\\
(\lambda\alpha.t_\alpha)^{\mathsf{M}}\,\gamma^2 &:\equiv \lambda\alpha.(t_\alpha)^{\mathsf{M}}\,\gamma^2\\
((\alpha \in T) \to b_\alpha)^{\mathsf{E}}\,\gamma^3 &:\equiv \lambda f\,\alpha.\,(b_\alpha)^{\mathsf{E}}\,\gamma^3\,(f\,\alpha)\\
(t\,\alpha)^{\mathsf{E}}\,\gamma^3 &:\equiv \mathsf{ap}\,(\lambda f.f\,\alpha)\,(t^{\mathsf{E}}\,\gamma^3)\\
(\lambda\alpha.t_\alpha)^{\mathsf{E}}\,\gamma^3 &:\equiv \mathsf{funext}\,(\lambda\alpha.\,(t_\alpha)^{\mathsf{E}}\,\gamma^3)
\end{aligned}$$

Only the $-^{\mathsf{E}}$ definitions are notably different. Since $(\alpha \in T) \to b_\alpha$ is in U, $((\alpha \in T) \to b_\alpha)^{\mathsf{E}}$ yields an eliminator function, and $(t\,\alpha)^{\mathsf{E}}$ yields a witness for a β-rule with type $(b_\alpha)^{\mathsf{E}}\,\gamma^3\,(t^{\mathsf{C}}\,\gamma\,\alpha) = t^{\mathsf{M}}\,\gamma^2\,\alpha$. Since $t^{\mathsf{E}}\,\gamma^3 \in (\lambda\alpha.\,(b_\alpha)^{\mathsf{E}}\,\gamma^3\,(t^{\mathsf{C}}\,\gamma\,\alpha)) =$

$t^{\mathsf{M}}\,\gamma^2$, we need to apply both sides of the equation to α with $\mathsf{ap}\,(\lambda f.f\,\alpha)$ to get the desired type. In the notation of homotopy type theory, the definition could be given as $\mathsf{happly}\,(t^{\mathsf{E}}\,\gamma^3)\,\alpha$.

For $(\lambda\alpha.t_\alpha)^{\mathsf{E}}$, the type of the right hand side is $(\lambda\alpha.\,(b_\alpha)^{\mathsf{E}}\,\gamma^3\,((t_\alpha)^{\mathsf{C}}\,\gamma)) = (\lambda\alpha.\,(t_\alpha)^{\mathsf{M}}\,\gamma^2)$. For any α, $(t_\alpha)^{\mathsf{E}}\,\gamma^3$ has type $(b_\alpha)^{\mathsf{E}}\,\gamma^3\,((t_\alpha)^{\mathsf{C}}\,\gamma) = (t_\alpha)^{\mathsf{M}}\,\gamma^2$, so we get the right witness by applying funext.

For W_{SP}, this implementation yields the elimination principle previously presented in section 3. We also present the example of indexed W-types (with codes, $-^{\mathsf{C}}$, $-^{\mathsf{M}}$, and $-^{\mathsf{E}}$) in appendix B. For a more complex infinitary example, see the QIIT definition of Cauchy reals in [21, p. 383]. It can be directly translated to a code; we omit reproducing the definitions here.

10 Towards Higher Inductive Inductive Types

The theory of codes is able to represent higher inductive inductive types (HIITs) as shown by the examples of set truncation and the torus in section 4. However, the metatheory has uniqueness of identity proofs (UIP), so the types returned by $-^{\mathsf{C}}$ are sets, i. e. all of their higher equalities are trivial. The question arises whether it is possible to remove UIP from the metatheory and work in full homotopy type theory [21].

If the metatheory lacks UIP, the QIIT defining the syntax needs to be set-truncated, i. e. we would need constructors expressing that in the syntax, all equalities of the same type are equal to each other. If we did not set-truncate, types and terms which only differ in witnesses for conversion proofs would be distinct.

However, with set-truncation, we can only eliminate from the syntax to types which are sets themselves — which the metatheoretic universe is not. In [3, section 6], this problem is fixed with an inductive recursive universe which can be proven to be a set. This solution does not work in our case, as we would like to have higher inductive types in the metatheory. We hope that using intermediate types which can be proven to be sets, we might be able to eliminate from the syntax of type theory into a type which is not a set [14]. A candidate for such an intermediate type could be a syntax of normal forms with decidable equality.

Another solution might be defining the operations $-^{\mathsf{C}}$, $-^{\mathsf{M}}$, $-^{\mathsf{E}}$ as syntactic translations [7] instead of targeting the metatheory, as in this case the target syntax would be a set again. In fact, the $-^{\mathsf{M}}$ operation was first described as a syntactic translation [6]. However, in this case we would have a problem defining $-^{\mathsf{E}}$: it only preserves the β rules in the theory of codes up to internal propositional equality, but not definitional equality. This is not a problem when targeting the metatheory, because there is no way to talk about definitional equalities there. This could be solved by leaving out the λ and refl constructors from the theory of codes together with the β rules. In this case we would need a different eliminator for equality, possibly a generalisation of transport (tr). However, it is not clear what would be the best way of representing non-inductive functions in this case.

A different way of solving the coherence issue would be using two-level type theory [4] where the syntax of the theory of codes could be given in the strict layer of the theory.

11 Related work and conclusions

Quotient types [13] are precursors of higher inductive types (HITs). The notion of HIT first appeared in [21], however only through examples and without a general definition. Sojakova [22] defines a subset of HITs called W-suspensions by an internal coding scheme similar to W-types. She proves that the induction principle is equivalent to homotopy initiality. Basold et al. [5] define an external syntactic scheme for higher inductive types with only 0-constructors and compute the types of elimination principles. In [25] a semantics is given for the same class of HITs but with no recursive equality constructors. Lumsdaine and Shulman give a general specification on when a model of type theory supports higher inductive types [17]. They introduce the notion of cell monad with parameters and characterise the class of models which have intial algebras for a cell monad with parameters. Kraus [15] and van Doorn [9] construct propositional truncation as a sequential colimit. The schemes mentioned so far do not support higher inductive inductive types.

Inductive inductive types were introduced in [19] together with an internal coding scheme. The closest to our work is Altenkirch et al's paper [2] which gives a categorical specification of QIITs. In this work, the definitions of sorts and constructors are given separately. Sorts are specified as a list of functors into Set where the domain of the functor is a category constructed from results of the previous functors, thus encoding dependencies of later sorts on previous ones. The constructors are specified mutually with their category of algebras and underlying carrier functor. The specification supports set-level equality constructors. From a specification of a QIIT they derive the type of the eliminator and show that this corresponds to initiality.

Dependencies in higher inductive inductive definitions can be very complex. We tackle this problem with a well-known method of describing intricate dependencies: the syntax of type theory. We have to limit our type formers to only allow strictly positive definitions, but these restrictions are the only things that a type theorist has to learn to understand our codes. Our encoding is also *direct* in the sense that the types of constructors and eliminators are exactly as required and not merely up to isomorphisms. The price we pay is that we need to handle an internal QIIT syntax of type theory. In the future, we would like to close the loop by defining a type theory supporting QIITs in which the same type theory can be internalised. Developing semantics for and proving existence of our encoded types is also future work.

Acknowledgements. We would like to thank Balázs Kőműves, Péter Diviánszky, Paolo Capriotti, Thorsten Altenkirch, Nicolas Tabareau and Simon Boulier for discussions and work related to this paper.

References

1. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers — constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.
2. Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. *CoRR*, abs/1612.02346, 2016.
3. Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016.
4. Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. Two-level type theory and applications. *ArXiv e-prints*, may 2017.
5. Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23(1):63–88, jan 2017.
6. Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012.
7. Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 182–194, New York, NY, USA, 2017. ACM.
8. James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. *ACM Sigplan Notices*, 45(9):3–14, 2010.
9. Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 122–129, New York, NY, USA, 2016. ACM.
10. Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.
11. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:525–549, 2000.
12. Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.
13. Martin Hofmann. *Extensional concepts in intensional type theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.
14. Nicolai Kraus. *Truncation Levels in Homotopy Type Theory.* PhD thesis, University of Nottingham, 2015.
15. 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.
16. Marc Lasson. Canonicity of weak -groupoid laws using parametricity theory. *Electronic Notes in Theoretical Computer Science*, 308:229 – 244, 2014.
17. Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2017.
18. Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.
19. Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

20. Christine Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.
21. The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.
22. Kristina Sojakova. Higher inductive types as homotopy-initial algebras. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 31–42, New York, NY, USA, 2015. ACM.
23. Thomas Streicher. *Investigations into intensional type theory.* `http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf`, 1993.
24. The Agda development team. Agda, 2015.
25. Niels van der Weide. *Higher Inductive Types.* PhD thesis, Radboud University, Nijmegen, 2016. Masters thesis.

A Definition of $-^{\mathsf{C}}$

For the theory of codes:

$$
\begin{aligned}
&.^{\mathsf{C}} &&:\equiv \top \\
&(\Gamma, x : A)^{\mathsf{C}} &&:\equiv (\gamma \in \Gamma^{\mathsf{C}}) \times A^{\mathsf{C}}\,\gamma \\
&x^{\mathsf{C}}\,\gamma &&:\equiv x^{\text{th}} \text{ component in } \gamma \\
&\mathsf{U}^{\mathsf{C}}\,\gamma &&:\equiv \mathsf{Set}_0 \\
&(\underline{a})^{\mathsf{C}}\,\gamma &&:\equiv a^{\mathsf{C}}\,\gamma \\
&((x : a) \to B)^{\mathsf{C}}\,\gamma &&:\equiv (\alpha \in a^{\mathsf{C}}\,\gamma) \to B^{\mathsf{C}}\,(\gamma, \alpha) \\
&(t\,u)^{\mathsf{C}}\,\gamma &&:\equiv (t^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{C}}\,\gamma) \\
&(\lambda x.t)^{\mathsf{C}}\,\gamma &&:\equiv \lambda\alpha.t^{\mathsf{C}}\,(\gamma, \alpha) \\
&((\alpha \in T) \to B_\alpha)^{\mathsf{C}}\,\gamma &&:\equiv (\alpha \in T) \to (B_\alpha)^{\mathsf{C}}\,\gamma \\
&(t\,\alpha)^{\mathsf{C}}\,\gamma &&:\equiv (t^{\mathsf{C}}\,\gamma)\,\alpha \\
&(\lambda\alpha.t_\alpha)^{\mathsf{C}}\,\gamma &&:\equiv \lambda\alpha.(t_\alpha)^{\mathsf{C}}\,\gamma \\
&(\mathsf{Id}_a\,t\,u)^{\mathsf{C}}\,\gamma &&:\equiv t^{\mathsf{C}}\,\gamma = u^{\mathsf{C}}\,\gamma \\
&\mathsf{refl}^{\mathsf{C}}\,\gamma &&:\equiv \mathsf{refl} \\
&(\mathsf{J}_{a\,t\,(x.z.P)}\,pr\,u\,eq)^{\mathsf{C}}\,\gamma &&:\equiv \mathsf{J}_{(a^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{C}}\,\gamma)\,(\lambda x\,z.P^{\mathsf{C}}\,(\gamma,x,z))}\,(pr^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{C}}\,\gamma)\,(eq^{\mathsf{C}}\,\gamma)
\end{aligned}
$$

For small non-inductive functions as defined in section 9:

$$
\begin{aligned}
&((\alpha \in T) \to b_\alpha)^{\mathsf{C}}\,\gamma &&:\equiv (\alpha \in T) \to (b_\alpha)^{\mathsf{C}}\,\gamma \\
&(t\,\alpha)^{\mathsf{C}}\,\gamma &&:\equiv (t^{\mathsf{C}}\,\gamma)\,\alpha \\
&(\lambda\alpha.t_\alpha)^{\mathsf{C}}\,\gamma &&:\equiv \lambda\alpha.(t_\alpha)^{\mathsf{C}}\,\gamma
\end{aligned}
$$

B Indexed W-types

Suppose $I \in Set_0$, $S \in \mathsf{Set}_0$, $P \in S \to \mathsf{Set}_0$, $Out \in S \to I$ and $In \in (s \in S) \to P\,s \to I$. Then, the code for the corresponding indexed W-type is the following:

$$W :\equiv (\cdot,\ \ w : (i \in I) \to \mathsf{U},\ \ sup : (s \in S) \to ((p \in P) \to w\,(In\,s\,p)) \to \underline{w\,(Out\,s)})$$

We pick a universe level j for elimination. The interpretations of W are the following, omitting leading $\top$ components:

$$\begin{aligned}
&W^{\mathsf{C}} \equiv (w \in I \to \mathsf{Set}_0) \times ((s \in S) \to ((p \in P\,s) \to w\,(In\,s\,p)) \to w\,(Out\,s))\\
&W^{\mathsf{M}}\,(w, sup) \equiv (w^M \in (i \in I) \to w\,i \to Set_j)\\
&\qquad\qquad\qquad \times ((s \in S)(f \in (p \in P\,s) \to w\,(In\,s\,p))\\
&\qquad\qquad\qquad\quad \to ((p \in P\,s) \to w^M\,(In\,s\,p)\,(f\,p)) \to w^M\,(Out\,s)\,(sup\,s\,f))\\
&W^{\mathsf{E}}\,(w, sup)\,(w^M, sup^M) \equiv\\
&\qquad\qquad\qquad\quad (w^E \in (i \in I)(x \in w\,i) \to w^M\,i\,x)\\
&\qquad\qquad\qquad \times ((s \in S)(f \in (p \in P\,s) \to w\,(In\,s\,p))\\
&\qquad\qquad\qquad\quad \to w^E\,(Out\,s)\,(sup\ s\,f) = sup^M\ s\,f\,(\lambda p.\,w^E\,(In\,s\,p)\,(f\,p)))
\end{aligned}$$

The main difference to W-types is the extra indexing in $w : (i \in I) \to \mathsf{U}$. However, we only index with elements of a metatheoretic I, using a non-inductive function for the type of w, hence only non-inductive $t\,\alpha$ applications appear in types, and J does not appear in the $-^{\mathsf{E}}$ output.

C Definition of J^{M}

The short definition was $\mathsf{J}\,\big(\mathsf{J}\,(pr^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{C}}\,\gamma)\big)\,(eq^{\mathsf{M}}\,\gamma^2)$. The inner $\mathsf{J}\,(pr^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{C}}\,\gamma)$ is expanded to:

$$\begin{aligned}
&\mathsf{J}_{(a^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{C}}\,\gamma)}\\
&\quad{}_{(\lambda x\,z.P^{\mathsf{M}}\,(\gamma,x,z)\,(\gamma^{\mathsf{M}},\mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,z\,(t^{\mathsf{M}}\,\gamma^2),\mathsf{refl}))}\\
&\qquad\qquad{}_{(\mathsf{J}_{(a^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{C}}\,\gamma)\,(\lambda x\,z.P^{\mathsf{C}}\,(\gamma,x,z))}\,(pr^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{C}}\,\gamma)\,(eq^{\mathsf{C}}\,\gamma))}\\
&\quad (pr^{\mathsf{M}}\,\gamma^2)\,(u^{\mathsf{C}}\,\gamma)\,(eq^{\mathsf{C}}\,\gamma)\\
&\in P^{\mathsf{M}}\,(\gamma, u^{\mathsf{C}}\,\gamma, eq^{\mathsf{C}}\,\gamma)\,(\gamma^{\mathsf{M}}, \mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,(eq^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{M}}\,\gamma^2), \mathsf{refl})\\
&\qquad (\mathsf{J}_{(a^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{C}}\,\gamma)\,(\lambda x\,z.P^{\mathsf{C}}\,(\gamma,x,z))}\,(pr^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{C}}\,\gamma)\,(eq^{\mathsf{C}}\,\gamma))
\end{aligned}$$

The outer J applied to the short inner J form is expanded as follows:

$$\begin{aligned}
&\mathsf{J}_{(a^{\mathsf{M}}\,\gamma^2)\,(\mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,(eq^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{M}}\,\gamma^2))} \\
&\quad(\lambda x\,z.P^{\mathsf{M}}\,(\gamma,u^{\mathsf{C}}\,\gamma,eq^{\mathsf{C}}\,\gamma)\,(\gamma^{\mathsf{M}},x,z)) \\
&\qquad\quad(\mathsf{J}_{(a^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{C}}\,\gamma)\,(\lambda x\,z.P^{\mathsf{C}}\,(\gamma,x,z))}\,(pr^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{C}}\,\gamma)\,(eq^{\mathsf{C}}\,\gamma)) \\
&\big(\mathsf{J}\,(pr^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{C}}\,\gamma)\big)\,(u^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{M}}\,\gamma^2) \\
&\in P^{\mathsf{M}}\,(\gamma,u^{\mathsf{C}}\,\gamma,eq^{\mathsf{C}}\,\gamma)\,(\gamma^{\mathsf{M}},u^{\mathsf{M}}\,\gamma^2\,eq^{\mathsf{M}}\,\gamma^2) \\
&\qquad(\mathsf{J}_{(a^{\mathsf{C}}\,\gamma)\,(t^{\mathsf{C}}\,\gamma)\,(\lambda x\,z.P^{\mathsf{C}}\,(\gamma,x,z))}\,(pr^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{C}}\,\gamma)\,(eq^{\mathsf{C}}\,\gamma))
\end{aligned}$$

D Definition of $(t\,u)^{\mathsf{E}}$

The expanded definition of $(t\,u)^{\mathsf{E}}\,\gamma^3$ is

$$\begin{aligned}
&\mathsf{J}_{(a^{\mathsf{M}}\,\gamma^2\,(u^{\mathsf{C}}\,\gamma))\,(a^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma))} \\
&\quad(\lambda x\,z.B^{\mathsf{E}}\,(\gamma,u^{\mathsf{C}}\,\gamma)\,(\gamma^{\mathsf{M}},x)\,(\gamma^{E},z)\,(t^{\mathsf{C}}\,\gamma\,(u^{\mathsf{C}}\,\gamma))\,(t^{\mathsf{M}}\,\gamma^2\,(u^{\mathsf{C}}\,\gamma)\,x)) \\
&\quad(t^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma))\,(u^{\mathsf{M}}\,\gamma^2)\,(u^{\mathsf{E}}\,\gamma^3).
\end{aligned}$$

E Definition of $(\mathsf{Id}_a\,t\,u)^{E}$

The goal is the dashed equality below. We can witness it by composing $\mathsf{apd}\,(a^{\mathsf{E}}\,\gamma^3)\,e$ and the two β equalities on the sides.

$$\begin{tikzcd}
\mathsf{tr}_{a^{\mathsf{M}}\,\gamma^2}\,e\,(a^{\mathsf{E}}\,\gamma^3\,(t^{\mathsf{C}}\,\gamma)) \arrow[r, dash, "\mathsf{apd}\,(a^{\mathsf{E}}\,\gamma^3)\,e"] \arrow[d, dash, "t^{\mathsf{E}}\,\gamma^3"'] & a^{\mathsf{E}}\,\gamma^3\,(u^{\mathsf{C}}\,\gamma) \arrow[d, dash, "u^{\mathsf{E}}\,\gamma^3"] \\
\mathsf{tr}_{a^{\mathsf{M}}\,\gamma^2}\,e\,(t^{\mathsf{M}}\,\gamma^2) \arrow[r, dash, dashed] & u^{\mathsf{M}}\,\gamma^2
\end{tikzcd}$$

The expanded definition for $(\mathsf{Id}_a\,t\,u)^{\mathsf{E}}\,\gamma^3$ is

$$\begin{aligned}
&\lambda e.\,\mathsf{tr}_{(\lambda x.\mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,e\,x=u^{\mathsf{M}}\,\gamma^2)}\,(t^{\mathsf{E}}\,\gamma^3) \\
&\qquad\Big(\mathsf{tr}_{(\lambda x.\mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,e\,(t^{\mathsf{M}}\,\gamma^2)=x)}\,(u^{\mathsf{E}}\,\gamma^3)\big(\mathsf{apd}\,(a^{\mathsf{E}}\,\gamma^3)\,e\big)\Big).
\end{aligned}$$

F Definition of $(\mathsf{refl}_t)^{\mathsf{E}}$

The expanded definition of $(\mathsf{refl}_t)^{\mathsf{E}}\,\gamma^3$ is

$$\begin{aligned}
&\mathsf{J}_{(a^{\mathsf{M}}\,\gamma^2\,(t\,^{\mathsf{C}}\,\gamma))\,(a^{\mathsf{E}}\,\gamma^3\,(t\,^{\mathsf{C}}\,\gamma))\,(\lambda x\,z.\mathsf{tr}_z\,(\mathsf{tr}_z\,\mathsf{refl}_{a^{\mathsf{E}}\,\gamma^3\,(t^{\mathsf{C}}\,\gamma)})=\mathsf{refl}_x)} \\
&\quad(\mathsf{refl}_{a^{\mathsf{E}}\,\gamma^3\,(t^{\mathsf{C}}\,\gamma)})\,(t^{\mathsf{M}}\,\gamma^2)\,(t^{\mathsf{E}}\,\gamma^3).
\end{aligned}$$