A Syntax for Higher Inductive-Inductive Types

Ambrus Kaposi

Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
akaposi@inf.elte.hu
https://orcid.org/0000-0001-9897-8936

András Kovács

Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
kovacsandras@inf.elte.hu
https://orcid.org/0000-0002-6375-9781

Abstract

Higher inductive-inductive types (HIITs) generalise inductive types of dependent type theories in two directions. On the one hand they allow the simultaneous definition of multiple sorts that can be indexed over each other. On the other hand they support equality constructors, thus generalising higher inductive types of homotopy type theory. Examples that make use of both features are the Cauchy reals and the well-typed syntax of type theory where conversion rules are given as equality constructors. In this paper we propose a general definition of HIITs using a domain-specific type theory. A context in this small type theory encodes a HIIT by listing the type formation rules and constructors. The type of the elimination principle and its β-rules are computed from the context using a variant of the syntactic logical relation translation. We show that for indexed W-types and various examples of HIITs the computed elimination principles are the expected ones. Showing that the thus specified HIITs exist is left as future work. The type theory specifying HIITs was formalised in Agda together with the syntactic translations. A Haskell implementation converts the types of sorts and constructors into valid Agda code which postulates the elimination principles and computation rules.

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.

For example, the inductive type of natural numbers Nat is given by the constructors zero : Nat and suc : Nat $\to$ Nat. The eliminator corresponds to the usual notion of mathematical induction:

$$\mathsf{ElimNat} : (P : \mathsf{Nat} \to \mathsf{Type})(pz : P\,\mathsf{zero})\big(ps : (n : \mathsf{Nat}) \to P\,n \to P\,(\mathsf{suc}\,n)\big)(n : \mathsf{Nat}) \to P\,n$$

P is a family of types over natural numbers, which is called the *motive* of the eliminator. It can be viewed as a proof-relevant predicate on Nat. The arguments pz and ps are called the *methods* of the eliminator. The *target* of the eliminator is n and given methods for each constructor, the eliminator provides a witness of $P\,n$. Thus, if P holds for zero and suc preserves P, then P holds for all natural numbers. The behaviour of the eliminator is described by a *computation-rule* (β-rule) for each constructor:

$$\mathsf{ElimNat}\,P\,pz\,ps\,\mathsf{zero} \quad \equiv pz$$
$$\mathsf{ElimNat}\,P\,pz\,ps\,(\mathsf{suc}\,n) \equiv ps\,n\,(\mathsf{ElimNat}\,P\,pz\,ps\,n)$$

These express that the eliminator applied to a constructor expression reduces to an application of the corresponding induction method. From an operational point of view, ElimNat replaces all the zero and suc constructors with the given induction methods.

Dependent families of types can be defined in a similar way, for example vectors of A-elements $\mathsf{Vec}_A : \mathsf{Nat} \to \mathsf{Type}$ which are indexed by their length. Another generalisation of inductive types are mutual inductive types. However, these can be reduced to indexed families where indices classify constructors for each mutual type. Inductive-inductive types [23] are mutual definitions where this reduction does not work: here a type is defined together with a family indexed over it. An example is the following fragment of the well-typed syntax of type theory where the second sort Ty is indexed over the first sort Con, but constructors of Con also refer to Ty:

Con	: Type	sort of contexts
Ty	: Con $\to$ Type	sort of types given a context
$\bullet$	: Con	constructor for the empty context
$-\rhd-$	: $(\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$	constructor for context extension
U	: $(\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma$	constructor for a base type
Π	: $(\Gamma : \mathsf{Con})(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \rhd A) \to \mathsf{Ty}\,\Gamma$	constructor for dependent functions

There are two eliminators for this type: one for Con and one for Ty. Both take the same arguments: two motives ($P : \mathsf{Con} \to \mathsf{Type}$ and $Q : (\Gamma : \mathsf{Con}) \to P\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathsf{Type}$) and four methods (one for each constructor, we don't list these).

$$\mathsf{ElimCon} : (P : \dots)(Q : \dots) \to \dots \to (\Gamma : \mathsf{Con})\ \to P\,\Gamma$$
$$\mathsf{ElimTy}\ : (P : \dots)(Q : \dots) \to \dots \to (A : \mathsf{Ty}\,\Gamma) \to Q\,\Gamma\,(\mathsf{ElimCon}\,\Gamma)\,A$$

Note that the type of ElimTy refers to ElimCon, which is why this elimination principle is called recursive-recursive (analogously to the phrase "inductive-inductive").

Higher inductive types (HITs, [25, Chapter 6]) generalise inductive types in a different way: they allow constructors expressing equalities of elements of the type being defined. This enables, among others, the definition of types quotiented by a relation. For example, the type of integers Int can be given by a constructor pair : Nat $\to$ Nat $\to$ Int and an equality constructor $\mathsf{eq} : (a\,b\,c\,d : \mathsf{Nat}) \to a + d =_{\mathsf{Nat}} b + c \to \mathsf{pair}\,a\,b =_{\mathsf{Int}} \mathsf{pair}\,c\,d$ targetting an

equality of Int. The eliminator for Int expects a motive $P : \mathsf{Int} \to \mathsf{Type}$, a method for the pair constructor $p : (a\,b : \mathsf{Nat}) \to P\,(\mathsf{pair}\,a\,b)$ and a method for the equality constructor path. This method is a proof that given $e : a + d =_{\mathsf{Nat}} b + c$, $p\,a\,b$ is equal to $p\,c\,d$ (the types of which are equal by e). Thus the method for the equality constructor ensures that all functions defined from the quotiented type respect the relation. Since the integers are supposed to be a set (which means that any two equalities between the same two integers are equal), we would need an additional higher equality constructor $\mathsf{trunc} : (x\,y : \mathsf{Int}) \to (p\,q : x =_{\mathsf{Int}} y) \to p =_{x =_{\mathsf{Int}} y} q$. HITs allow equality constructors at any level. With the view of types as spaces in mind, point constructors add points to the space, equality constructors add paths and higher constructors add homotopies between paths.

Not all constructor expressions make sense. For example [25, Example 6.13.1], given an $f : (X : \mathsf{Type}) \to X \to X$, suppose that an inductive type Ival is generated by the point constructors $\mathsf{a} : \mathsf{Ival}$, $\mathsf{b} : \mathsf{Ival}$ and a path constructor $\sigma : f\,\mathsf{Ival}\,\mathsf{a} =_{\mathsf{Ival}} f\,\mathsf{Ival}\,\mathsf{b}$. The eliminator for this type should take a motive $P : \mathsf{Ival} \to \mathsf{Type}$, two methods $p_a : P\,\mathsf{a}$ and $p_b : P\,\mathsf{b}$, and a path connecting elements of $P\,(f\,\mathsf{Ival}\,\mathsf{a})$ and $P\,(f\,\mathsf{Ival}\,\mathsf{b})$. However it is not clear what these elements should be: we only have elements $p_a : P\,\mathsf{a}$ and $p_b : P\,\mathsf{b}$, and there is no way in general to transform these to have types $P\,(f\,\mathsf{Ival}\,\mathsf{a})$ and $P\,(f\,\mathsf{Ival}\,\mathsf{b})$.

Another invalid example is an inductive type Neg with a constructor $\mathsf{con} : (\mathsf{Neg} \to \bot) \to \mathsf{Neg}$ where $\bot$ is the empty type. An eliminator for this type should (at least) yield a projection function $\mathsf{proj} : \mathsf{Neg} \to (\mathsf{Neg} \to \bot)$. Given this, we can define $u :\equiv \mathsf{con}\,(\lambda x.\mathsf{proj}\,x\,x) : \mathsf{Neg}$ and then derive $\bot$ by $\mathsf{proj}\,u\,u$. The existence of Neg would make the type theory inconsistent. A common restriction to avoid such situations is *strict positivity*. It means that the type being defined cannot occur on the left hand side of a function arrow in a parameter of a constructor. This excludes the above constructor con.

In this paper we propose a general syntax for higher inductive-inductive types (HIITs) which includes the above positive examples and excludes the negative ones. Our syntax for HIITs allows any number of inductive-inductive sorts, possibly infinitary higher constructors of any dimension and restricts constructors to strictly positive ones. It also allows free usage of J and refl in HIIT specifications. We also show how to derive the types of the eliminators and computation rules from the type formation rules and constructors.

The core idea is to represent HIIT specifications as contexts in a domain-specific type theory which we call the *theory of codes*. A context in this theory can be seen as a *code* for a HIIT, similarly to how a container [1] can be seen as a code for a simple inductive type. Type formers in the theory of codes are restricted in order to enforce strict positivity. For example, natural numbers are defined as the three-element context

$$Nat : \mathsf{U}, \quad zero : \underline{Nat}, \quad suc : Nat \to \underline{Nat}$$

where Nat, $zero$ and suc are simply variable names, and underlining denotes El (decoding) for the Tarski-style universe U.

We use a variant of Bernardy et al.'s logical predicate translation [8] to derive the types of motives and methods, and a logical relation translation to derive the types of the eliminators and computation rules. The target of these translations is a type theory with a predicative hierarchy of Russell-style universes closed under Π, Σ, the equality (identity) type $- = -$ and the unit type $\top$. The source type theory is the target type theory extended with rules for the theory of codes.

To our knowledge, this is the first proposal for a definition of HIITs. Proving the existence of the HIITs thus specified is left as future work.

1.1   Overview of the paper

We start by describing the target type theory in Section 2. In Section 3, we define the source type theory. The source type theory is the target theory extended with the theory of codes, i.e. the rules to describe HIITs. We also provide several examples of HIIT definitions. In Section 4 we define three syntactic translations from the source to the target theory, each depending on the previous one: first, we compute the types of type formation rules and constructors (Section 4.1); then, assuming the constructors exist, we compute the types of motives and methods (Section 4.2); finally we compute the types of the eliminators together with their computation rules (Section 4.3). To illustrate these operations, we show how they compute on a few example codes: natural numbers, the circle, indexed W-types and the two-dimensional sphere (Appendix A). In Section 5 we add the pieces together by specifying what it means for the target type theory to support HIITs. Section 6 describes the formalisation and a Haskell implementation. We conclude in Section 7.

1.2   Related work

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. 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. Our development uses the former approach.

External schemes for inductive families are given in [13, 24], for inductive-recursive types in [14]. A symmetric scheme for both inductive and coinductive types is given in [5]. Basold et al. [6] define an external syntactic scheme for higher inductive types with only 0-constructors and compute the types of elimination principles. In [27] a semantics is given for the same class of HITs but with no recursive equality constructors. Dybjer and Moeneclaey define a syntactic scheme for finitary HITs and show their existence in a groupoid model [15].

Internal 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]. Morris and Altenkirch [22] extend the notion of container to that of indexed container which specifies indexed inductive types. Codes for inductive-recursive types are given in [16]. Inductive-inductive types were introduced by Forsberg together with an internal coding scheme [23]. Sojakova [26] 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.

Quotient types [17] are precursors of higher inductive types (HITs). The notion of HIT first appeared in [25], however only through examples and without a general definition. Lumsdaine and Shulman give a general specification of models of type theory supporting higher inductive types [21]. 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 [19] and Van Doorn [12] construct propositional truncation as a sequential colimit. The schemes mentioned so far do not support higher inductive-inductive types.

The closest to our work is the article of Altenkirch et al. [2] which gives a categorical specification of quotient inductive-inductive types (QIITs), i.e. set-truncated higher inductive-inductive types. 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.

The logical predicate syntactic translation was introduced by Bernardy et al. [8]. The idea that a context can be seen as a definition of an inductive type and the logical predicate translation can be used to derive the types of motives and methods was described in [3, Section 5.3]. Logical relations are used to derive the computation rules in [18, Section 4.3], however only for closed QIITs. Syntactic translations in the context of the calculus of inductive constructions are discussed in [10]. Logical relations and parametricity can also be used to justify the *existence* of inductive types in a type theory with an impredicative universe [4]. In contrast, we only use logical relations to *describe* HIITs.

2 Target type theory

In this section we describe the target type theory. It is the target of our translations, and it also serves as a source of constants which are external to the HIIT being defined. It has Russell-style universes, Π, Σ, equality (identity) and unit type. Our notation is close to Agda's: we use named variables, terms are identified up to α-conversion, substitution and weakening are implicit. To distinguish notation from the theory of codes described in the next section, we write term formers and metavariables in brick red colour. We have the following judgement kinds.

$\vdash \Gamma$ $\qquad\qquad$ Γ is a valid target context

$\Gamma \vdash t : A$ $\qquad$ the target term t has target type A in target context Γ

We only describe the target type theory in informal English instead of writing down all the rules, since they are standard. See [25, Appendix A.2] for a formal treatment.

Context extension is written $\Gamma, x : A$. We have a cumulative hierarchy of universes Type_i.

Dependent function space is denoted $(x : A) \to B$. We write $A \to B$ if B does not depend on x, and $\to$ associates to the right, $(x : A)(y : B) \to C$ abbreviates $(x : A) \to (y : B) \to C$ and $(x\,y : A) \to B$ abbreviates $(x : A)(y : A) \to B$. We write $\lambda x.t$ for abstraction and $t\,u$ for left-associative application.

$(x : A) \times B$ stands for Σ types, $A \times B$ for the non-dependent version and $\times$ associates to the left. The constructor for Σ types is denoted (t, u) with eliminators proj_1 and proj_2. Both Π and Σ have definitional β and η rules.

The equality (identity) type for a type A and elements $t : A$, $u : A$ is denoted $t =_A u$ and comes with the constructor refl_t and eliminator J with definitional β-rule. The notation is $\mathsf{J}_{A\,t\,P}\,pr\,_u\,eq$ for $t : A$, $P : (x : A) \to t =_A x \to \mathsf{Type}_i$, $pr : P\,t\,\mathsf{refl}$ and $eq : t =_A u$. Sometimes we omit parameters in subscripts.

We will use the following functions defined using J in the standard way. We write $\mathsf{tr}_P\,e\,t : P\,v$ for transport of $t : P\,u$ along $e : u = v$. We write $\mathsf{ap}\,f\,e : f\,u = f\,v$ for $f : A \to B$ and $e : u = v$, $\mathsf{apd}\,f\,e : \mathsf{tr}_P\,e\,(f\,u) = f\,v$ for $f : (x : A) \to B$ and $e : u = v$.

The unit type is denoted $\top$ with constructor tt.

3 Source type theory

The source type theory is the target type theory extended with the following judgement kinds.

$\Gamma \vdash \Delta$ $\qquad\qquad$ Δ is a context in the target context Γ

$\Gamma; \Delta \vdash A$ $\qquad\quad$ A is a type in context Δ and target context Γ

$\Gamma; \Delta \vdash t : A$ $\qquad$ t is a term of type A in context Δ and target context Γ

(1) Contexts and variables

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

(2) Universe

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

(3) Inductive parameters

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

(4) Equality

$$\frac{\Gamma; \Delta \vdash a : \mathsf{U} \qquad \Gamma; \Delta \vdash t : \underline{a} \qquad \Gamma; \Delta \vdash u : \underline{a}}{\Gamma; \Delta \vdash t =_a u : \mathsf{U}} \qquad \frac{\Gamma; \Delta \vdash t : \underline{a}}{\Gamma; \Delta \vdash \mathsf{refl} : \underline{t =_a t}}$$

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

$$\frac{\Gamma; \Delta \vdash t : \underline{a} \qquad \Gamma; \Delta, x : \underline{a}, z : \underline{t =_a x} \vdash p : \mathsf{U} \qquad \Gamma; \Delta \vdash pr : \underline{p[x \mapsto t, z \mapsto \mathsf{refl}]}}{\Gamma; \Delta \vdash \mathsf{J}\beta_{a\,t\,(x.z.p)}\, pr : \underline{(\mathsf{J}_{a\,t\,(x.z.p)}\, pr\,{}_t\, \mathsf{refl}) =_{p[x \mapsto t, z \mapsto \mathsf{refl}]} pr}}$$

(5) Non-inductive parameters

$$\frac{\Gamma \vdash A : \mathsf{Type}_0 \qquad \Gamma; \vdash \Delta \qquad (\Gamma, x : A); \Delta \vdash B}{\Gamma; \Delta \vdash (x : A) \to B}$$

$$\frac{\Gamma; \Delta \vdash t : (x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma; \Delta \vdash t\,u : B[x \mapsto u]}$$

(6) Infinitary parameters

$$\frac{\Gamma \vdash A : \mathsf{Type}_0 \qquad \Gamma; \vdash \Delta \qquad (\Gamma, x : A); \Delta \vdash b : \mathsf{U}}{\Gamma; \Delta \vdash (x : A) \to b : \mathsf{U}}$$

$$\frac{\Gamma; \Delta \vdash t : \underline{(x : A) \to b} \qquad \Gamma \vdash u : A}{\Gamma; \Delta \vdash t\,u : \underline{b[x \mapsto u]}}$$

■ **Figure 1** The theory of HIIT codes (part of the source type theory). Substitution and weakening are implicit, we assume fresh names everywhere and consider α-convertible terms equal. The $\Gamma;$ assumptions are not used or changed in parts (1)–(4).

We name the subset of rules of the source theory which derives these judgements *the theory of codes*. The derivation rules are listed in figure 1. A context Δ for which $\Gamma \vdash \Delta$ can be derived represents a specification of a HIIT.

Although every judgement is valid up to a context in the target type theory, note that none of the rules in (1)–(4) depend on or change these assumptions, so they can be safely ignored until part (5). We explain the rules in order.

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

(2) There is a universe U, with decoding written as an underline (usually El in the literature). Type formation rules will target U. With this part of the syntax we can already define contexts specifying the empty type, unit type and booleans:

$$\cdot,\ Empty : \mathsf{U} \qquad \cdot,\ Unit : \mathsf{U},\ tt : \underline{Unit} \qquad \cdot,\ Bool : \mathsf{U},\ true : \underline{Bool},\ false : \underline{Bool}$$

(3) We have a function space with small domain and large codomain. This can be used to add inductive arguments to type formation rules and constructors. As U is not closed under this function space, these function types cannot (recursively) appear in inductive arguments, which ensures strict positivity. When the codomain does not depend on the domain, $a \to B$ can be written instead of $(x : a) \to B$.

Now we can specify the natural numbers as a context:

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

We can also encode inductive-inductive definitions such as the fragment of the well-typed syntax of a type theory mentioned in the introduction:

$$\cdot,\ Con : \mathsf{U},\ Ty : Con \to \mathsf{U},\ \bullet : \underline{Con},\ -\rhd- : (\Delta : Con) \to Ty\,\Delta \to \underline{Con},$$
$$U : (\Delta : Con) \to \underline{Ty\,\Delta},\ \Pi : (\Delta : Con)(A : Ty\,\Delta)(B : Ty\,(\Delta \rhd A)) \to \underline{Ty\,\Delta}$$

(4) U is closed under the equality type, with eliminator J and a weak (non-definitional) β-rule. Weakness is required because the syntactic translation $-^{\mathsf{E}}$ defined in Section 4.3 does not preserve this β-rule strictly. Adding equality to the theory of codes allows higher constructors and inductive equality parameters as well. We can now define the circle HIT as the following context:

$$\cdot,\ S^1 : \mathsf{U},\ base : \underline{S^1},\ loop : \underline{base =_{S^1} base}$$

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: given $p : \underline{t =_a u}$ and $q : \underline{u =_a v}$, $p \bullet q$ abbreviates $\mathsf{J}_{a\,u\,x.z.(t=x)}\,p\,_v\,q : \underline{t =_a v}$. The torus is given as follows.

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

With the equality type at hand, we can define a full well-typed syntax of type theory as given e.g. in [3] as an inductive type (see the examples in the formalisation described in Section 6).

So far we were only able to define closed HIITs, which excludes lists of a given type or the integers as given in the introduction. This is where we need the target theory to be included in the source theory. A context Δ for which $\Gamma \vdash \Delta$ holds can be seen as a specification of an inductive type which depends on Γ. In the case of lists, Γ will be $A : \mathsf{Type}_0$. In the case of integers, we need $Nat : \mathsf{Type}_0$ and $-+- : Nat \to Nat \to Nat$ from Γ.

(5) We have a function space where the domain is a type in the target theory. We distinguish it from (3) by using red brick : instead of : in the domain specification. We specify lists and the integers as follows.

$$A : \mathsf{Type}_0 \vdash \; \cdot, \; List : \mathsf{U}, \; nil : \underline{List}, \; cons : (x : A) \to List \to \underline{List}$$

$$\Gamma \qquad \vdash \; \cdot, \; Int : \mathsf{U}, \; pair : (x\,y : Nat) \to \underline{Int},$$
$$eq : (a\,b\,c\,d : Nat)(p : a{+}d =_{Nat} b{+}c) \to \underline{pair\,a\,b =_{Int} \mathsf{pair}\,c\,d},$$
$$trunc : (xy : Int)(p\,q : a =_{Int} b) \to \underline{p =_{x=_{Int}y} q}$$

In the case of integers, Γ is $Nat : \mathsf{Type}_0, -{+}- : Nat \to Nat \to Nat$, or alternatively, we could require natural numbers in the target theory. As another example, propositional truncation for a type A is specified as follows.

$$A : \mathsf{Type}_0 \vdash \; \cdot, \; tr : \mathsf{U}, \; emb : (x : A) \to \underline{tr}, \; eq : (x\,y : tr) \to \underline{x =_{tr} y}$$

The smallness of A is required in (5). It is possible to generalize the syntax of HIITs to arbitrary universe levels, but it is not essential to the current development. Note that the (5) function space preserves strict positivity, since in the target theory there is no way to recursively refer to the inductive type *being defined*. The situation is analogous to the case of W-types [1], where shapes and positions can contain arbitrary types but they cannot recursively refer to the W-type being defined.

(6) U is also closed under a function space where the domain is a target theory type and the codomain is a small source theory type. We overload the application notation for non-inductive parameters, as it is usually clear from context which application is meant. The rules allow types with infinitary constructors, for example trees containing A-elements at the leaves and branching by B (which could be an infinite type):

$$A : \mathsf{Type}_0, B : \mathsf{Type}_0 \vdash \; \cdot, \; T : \mathsf{U}, \; leaf : (x : A) \to \underline{T}, \; node : ((x : B) \to T) \to \underline{T}$$

Here, $leaf$ has a function type (5) and $node$ has a function type (3) with a function type (6) in the domain. More generally, we can define W-types [1] as follows. S describes the "shapes" of the constructors and P the "positions" where recursive arguments can appear.

$$S : \mathsf{Type}_0, P : S \to \mathsf{Type}_0 \vdash \; \cdot, \; W : \mathsf{U}, \; sup : (s : S) \to ((p : P\,s) \to W) \to \underline{W}$$

For a more complex infinitary example, see the definition of Cauchy reals in [25, Definition 11.3.2]. It can be also found as an example file in our Haskell implementation.

The invalid examples Ival and Neg cannot be encoded by the theory of codes. For Ival, we can go as far as

$$\cdot, \; Ival : \mathsf{U}, \; a : \underline{Ival}, \; b : \underline{Ival}, \; \sigma : \underline{? =_{Ival} ?}.$$

The first argument of the function $f : (X : \mathsf{Type}) \to X \to X$ is a target theory type, but we only have $Ival : \mathsf{U}$ in the theory of codes. Neg cannot be typed because the first parameter of the constructor con is a function from a small type to a target theory type, and no such functions can be formed.

4 A syntactic translation from the source to the target theory

An inductive type is specified by a context in the theory of codes defined in the previous section. In this section we define the $-^{\mathsf{C}}$, $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ operations, which work as follows on the code for natural numbers.

$$(\cdot,\, Nat : \mathsf{U},\, zero : \underline{Nat},\, suc : Nat \to \underline{Nat})^{\mathsf{C}}$$
$$\equiv \top \times (n : \mathsf{Type}_0) \times (z : n) \times (n \to n)$$
$$(\cdot,\, Nat : \mathsf{U},\, zero : \underline{Nat},\, suc : Nat \to \underline{Nat})^{\mathsf{M}}\,(\mathsf{tt}, n, z, s)$$
$$\equiv \top \times (n^M : n \to \mathsf{Type}_i) \times (z^M : n^M\, z) \times \big((x : n) \to n^M\, x \to n^M\,(s\, x)\big)$$
$$(\cdot,\, Nat : \mathsf{U},\, zero : \underline{Nat},\, suc : Nat \to \underline{Nat})^{\mathsf{E}}\,(\mathsf{tt}, n, z, s)\,(\mathsf{tt}, n^M, z^M, s^M)$$
$$\equiv \top \times \big(n^E : (x : n) \to n^M\, x\big) \times (z^E : n^E\, z = z^M) \times \big((x : n) \to n^E\,(s\, x) = n^M\, x\,(n^E\, x)\big)$$

The brick red coloured result of $-^{\mathsf{C}}$ gives the types of the type formation rule and constructors as an iterated Σ-type. Assuming the existence of constructors, $-^{\mathsf{M}}$ returns the types of motives and methods. Assuming the existence of constructors and the corresponding motives and methods, $-^{\mathsf{E}}$ returns the types of the eliminator and computation rules as target theory equalities $=$. The notation above denotes *left*-nested iterated Σ-types. A more precise presentation would replace each variable with a projection from the preceding Σ-type. We use this notation in order to reduce visual clutter.

Each context entry in the theory of codes specifies a type formation rule or a constructor. In general, the last component of a type in a context entry is of three possible forms: it is either U, $\underline{a}$ for some neutral a, or $\underline{t =_a u}$. The following table summarizes the results of the various translations in the mentioned three cases:

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

The $-^{\mathsf{C}}$, $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ operations are defined by induction on the derivations of the source syntax. The operations are identity on derivations of target contexts and target terms (of the forms $\vdash \Gamma$ and $\Gamma \vdash t : A$) and derive target terms from theory of codes contexts, types and terms (of the forms $\Gamma \vdash \Delta$, $\Gamma; \Delta \vdash A$ and $\Gamma; \Delta \vdash t : A$, respectively). We only present the non-identity parts with pattern matching notation, describing how a context, type or a term in the theory of codes is converted to a term in the target theory.

All three operations respect definitional equality. This amounts to preserving equalities of the substitution calculus, as there are no β-rules introduced in the theory of codes.

4.1 Type formation rules and constructors

Given a context in the theory of codes, $-^{\mathsf{C}}$ returns the types of type formation rules and constructors as an iterated Σ-type in the target theory. It is specified as follows.

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

Given a context depending on the target context Γ, $-^{\mathsf{C}}$ returns a type in Γ. Given a type in context Δ it returns a family over Δ^{C}. A term is interpreted by a dependent function in the

target theory. The implementation of $-^{\mathsf{C}}$ is essentially the standard model, where everything is interpreted by its brick red counterpart, except for contexts which are interpreted as iterated Σ-types.

$$
\begin{aligned}
\cdot^{\mathsf{C}} &\; :\equiv \top \\
(\Delta, x : A)^{\mathsf{C}} &\; :\equiv (\gamma : \Delta^{\mathsf{C}}) \times A^{\mathsf{C}} \, \gamma \\
x^{\mathsf{C}} \, \gamma &\; :\equiv x^{\mathrm{th}} \text{ component in } \gamma \\
\mathsf{U}^{\mathsf{C}} \, \gamma &\; :\equiv \mathsf{Type}_0 \\
(\underline{a})^{\mathsf{C}} \, \gamma &\; :\equiv a^{\mathsf{C}} \, \gamma \\
((x : a) \to B)^{\mathsf{C}} \, \gamma &\; :\equiv (x : a^{\mathsf{C}} \, \gamma) \to B^{\mathsf{C}} \, (\gamma, x) \\
(t \, u)^{\mathsf{C}} \, \gamma &\; :\equiv (t^{\mathsf{C}} \, \gamma) \, (u^{\mathsf{C}} \, \gamma) \\
((x : A) \to B)^{\mathsf{C}} \, \gamma &\; :\equiv (x : A) \to B^{\mathsf{C}} \, \gamma \\
(t \, u)^{\mathsf{C}} \, \gamma &\; :\equiv (t^{\mathsf{C}} \, \gamma) \, u \\
(t =_a u)^{\mathsf{C}} \, \gamma &\; :\equiv t^{\mathsf{C}} \, \gamma = u^{\mathsf{C}} \, \gamma \\
\mathsf{refl}^{\mathsf{C}} \, \gamma &\; :\equiv \mathsf{refl} \\
(\mathsf{J}_{a \, t \, (x.z.p)} \, pr \, u \, eq)^{\mathsf{C}} \, \gamma &\; :\equiv \mathsf{J}_{(a^{\mathsf{C}} \, \gamma) \, (t^{\mathsf{C}} \, \gamma) \, (\lambda x \, z. p^{\mathsf{C}} \, (\gamma, x, z))} \, (pr^{\mathsf{C}} \, \gamma) \, (u^{\mathsf{C}} \, \gamma) \, (eq^{\mathsf{C}} \, \gamma) \\
(\mathsf{J}\beta_{a \, t \, (x.z.p)} \, pr)^{\mathsf{C}} \, \gamma &\; :\equiv \mathsf{refl} \\
((x : A) \to b)^{\mathsf{C}} \, \gamma &\; :\equiv (x : A) \to b^{\mathsf{C}} \, \gamma \\
(t \, u)^{\mathsf{C}} \, \gamma &\; :\equiv (t^{\mathsf{C}} \, \gamma) \, u
\end{aligned}
$$

For example, $-^{\mathsf{C}}$ acts as follows on the topological circle:

$$(\cdot, S^1 : \mathsf{U}, b : \underline{S^1}, loop : \underline{b = b})^{\mathsf{C}} \equiv \top \times (S^1 : \mathsf{Type}_0) \times (b : S^1) \times (loop : b = b)$$

The resulting left-nested Σ type could be written explicitly as below. We shall keep to the more readable notation from now on.

$$\left(x'' : \left(x' : (x : \top) \times \mathsf{Type}_0 \right) \times \mathsf{proj}_2 \, x' \right) \times (\mathsf{proj}_2 \, x'' = \mathsf{proj}_2 \, x'')$$

4.2   Motives and methods

Given a code for an inductive type and the constructors specified by the code, the operation $-^{\mathsf{M}}$ returns the induction motives and methods.

$-^{\mathsf{M}}$ is a variant of the unary logical predicate translation of Bernardy et al. [9]. We fix a level i for the universe we would like to eliminate into. For each context Δ, Δ^{M} is a predicate over the standard interpretation Δ^{C}. For a type $\Delta \vdash A$, A^{M} is a predicate over A^{C}, which also depends on $\gamma : \Delta^{\mathsf{C}}$ and a witness of $\Delta^{\mathsf{M}} \, \gamma$. All of these may refer to a target theory context Γ.

$$
\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta^{\mathsf{M}} : \Delta^{\mathsf{C}} \to \mathsf{Type}_{i+1}}
\qquad
\frac{\Gamma; \Delta \vdash A}{\Gamma \vdash A^{\mathsf{M}} : (\gamma : \Delta^{\mathsf{C}}) \to \Delta^{\mathsf{M}} \, \gamma \to A^{\mathsf{C}} \, \gamma \to \mathsf{Type}_{i+1}}
$$

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.

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

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.

$$
\begin{aligned}
\cdot^{\mathsf{M}}\,\gamma &:\equiv \top \\
(\Delta,\,x:A)^{\mathsf{M}}\,(\gamma,\,t) &:\equiv (\gamma^M:\Delta^{\mathsf{M}}\,\gamma)\times A^{\mathsf{M}}\,\gamma^2\,t \\
x^{\mathsf{M}}\,\gamma^2 &:\equiv x^{\text{th}}\text{ component in }\gamma^M \\
\mathsf{U}^{\mathsf{M}}\,\gamma^2\,A &:\equiv A\to\mathsf{Type}_i \\
(\underline{a})^{\mathsf{M}}\,\gamma^2\,t &:\equiv a^{\mathsf{M}}\,\gamma^2\,t \\
((x:a)\to B)^{\mathsf{M}}\,\gamma^2\,f &:\equiv (x:a^{\mathsf{C}}\,\gamma)(x^M:a^{\mathsf{M}}\,\gamma^2\,x)\to B^{\mathsf{M}}\,(\gamma,x)\,(\gamma^M,x^M)\,(f\,x) \\
(t\,u)^{\mathsf{M}}\,\gamma^2 &:\equiv (t^{\mathsf{M}}\,\gamma^2)\,(u^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{M}}\,\gamma^2) \\
((x:A)\to B)^{\mathsf{M}}\,\gamma^2\,f &:\equiv (x:A)\to B^{\mathsf{M}}\,\gamma^2\,(f\,x) \\
(t\,u)^{\mathsf{M}}\,\gamma^2 &:\equiv t^{\mathsf{M}}\,\gamma^2\,u \\
(t=_a u)^{\mathsf{M}}\,\gamma^2\,e &:\equiv \mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,e\,(t^{\mathsf{M}}\,\gamma^2)=u^{\mathsf{M}}\gamma^2 \\
(\mathsf{refl}_t)^{\mathsf{M}}\,\gamma^2 &:\equiv \mathsf{refl}_{(t^{\mathsf{M}}\,\gamma^2)} \\
(\mathsf{J}_{a\,t\,(x.z.p)}\,pr\,_u\,eq)^{\mathsf{M}}\,\gamma^2 &:\equiv \mathsf{J}\left(\mathsf{J}\,(pr^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{C}}\,\gamma)\right)(eq^{\mathsf{M}}\,\gamma^2) \\
(\mathsf{J}\beta_{a\,t\,(x.z.p)}\,pr)^{\mathsf{M}}\,\gamma^2 &:\equiv \mathsf{refl} \\
((x:A)\to b)^{\mathsf{M}}\,\gamma^2\,f &:\equiv (x:A)\to b^{\mathsf{M}}\,\gamma^2\,(f\,x) \\
(t\,u)^{\mathsf{M}}\,\gamma^2 &:\equiv t^{\mathsf{M}}\,\gamma^2\,u
\end{aligned}
$$

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 $A:\mathsf{U}^{\mathsf{C}}\,\gamma$ (with $\mathsf{U}^{\mathsf{C}}\,\gamma\equiv\mathsf{Type}_0$) returns the predicate space over A. The predicate for a type $\underline{a}$ is given by the predicate for a.

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

The predicate for the equality type $t=_a u$, for each $e:(t=_a u)^{\mathsf{C}}\,\gamma$, i.e. $e:t^{\mathsf{C}}\,\gamma=u^{\mathsf{C}}\,\gamma$, says that 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 target theory. The interpretation of J is given by a double J application, which definition is sourced from [20]. Here, we use a shortened J notation; we refer to the formalisation (Section 6) for the details.

Again, let us consider the circle example:

$$
\begin{aligned}
&(\cdot,\,S^1:\mathsf{U},\,b:\underline{S^1},\,loop:\underline{b=b})^{\mathsf{M}}\,(\mathsf{tt},\,S^1,\,b,\,loop) \\
\equiv\;&\top\times(S^{1M}:S^1\to\mathsf{Type}_i)\times(b^M:S^{1M}\,b)\times(loop^M:\mathsf{tr}_{\,S^{1M}}\,loop\,b^M=b^M)
\end{aligned}
$$

The inputs of $-^{\mathsf{M}}$ here are the code for the circle (the context in black) and a tuple of the type formation rule S^1, the constructor b and the equality constructor $loop$. It returns a family over the type S^1, an element of this family b^M at index b, and and an equality between b^M and b^M which lies over $loop$.

4.3 Eliminators and computation rules

The operation $-^{\mathsf{E}}$ yields eliminators and computation rules. It is a generalised binary logical relation translation where the type of the second parameter of 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.

$$\frac{\Gamma \vdash \Delta}{\Gamma; \Delta^\mathsf{E} : (\gamma : \Delta^\mathsf{C}) \to \Delta^\mathsf{M}\, \gamma \to \mathsf{Type}_i}$$

Types are interpreted as dependent binary relations which additionally depend on $(\gamma, \gamma^M, \gamma^E)$ interpretations of the context.

$$\frac{\Gamma; \Delta \vdash A}{\Gamma \vdash A^\mathsf{E} : (\gamma : \Delta^\mathsf{C})(\gamma^M : \Delta^\mathsf{M}\, \gamma)(\gamma^E : \Delta^\mathsf{E}\, \gamma\, \gamma^M)(x : A^\mathsf{C}\, \gamma) \to A^\mathsf{M}\, \gamma\, \gamma^M\, x \to \mathsf{Type}_i}$$

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

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

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

$$
\begin{aligned}
\cdot^\mathsf{E}\, \gamma\, \gamma^M &:\equiv \top \\
(\Delta,\, x : A)^\mathsf{E}\, (\gamma, t)\, (\gamma^M, t^M) &:\equiv (\gamma^E : \Delta^\mathsf{E}\gamma^2) \times A^\mathsf{E}\, \gamma^3\, t\, t^M \\
x^\mathsf{E}\, \gamma^3 &:\equiv x^{\text{th}} \text{ component in } \gamma^E \\
\mathsf{U}^\mathsf{E}\, \gamma^3\, A\, A^M &:\equiv (x : A) \to A^M\, x \\
(\underline{a})^\mathsf{E}\, \gamma^3\, t\, t^M &:\equiv a^\mathsf{E}\, \gamma^3\, t = t^M \\
((x : a) \to B)^\mathsf{E}\, \gamma^3\, f\, f^M &:\equiv (x : a^\mathsf{C}\, \gamma) \to B^\mathsf{E}\, (\gamma, x)\, (\gamma^M, a^\mathsf{E}\, \gamma^3\, x)\, (\gamma^E, \mathsf{refl}) \\
&\qquad\qquad\qquad\qquad (f\, x)\, \big(f^M\, x\, (a^\mathsf{E}\, \gamma^3\, x)\big) \\
(t\, u)^\mathsf{E}\, \gamma^3 &:\equiv \mathsf{J}\, (t^\mathsf{E}\, \gamma^3\, (u^\mathsf{C}\, \gamma))\, (u^\mathsf{E}\, \gamma^3) \\
((x : A) \to B)^\mathsf{E}\, \gamma^3\, f\, f^M &:\equiv (x : A) \to B^\mathsf{E}\, \gamma^3\, (f\, x)\, (f^M\, x) \\
(t\, u)^\mathsf{E}\, \gamma^3 &:\equiv t^\mathsf{E}\, \gamma^3\, u \\
(t =_a u)^\mathsf{E}\, \gamma^3\, e &:\equiv \mathsf{tr}\, (t^\mathsf{E}\, \gamma^3)\big(\mathsf{tr}\, (u^\mathsf{E}\, \gamma^3)\, (\mathsf{apd}\, (a^\mathsf{E}\, \gamma^3)\, e)\big) \\
(\mathsf{refl}_t)^\mathsf{E}\, \gamma^3 &:\equiv \mathsf{J}\, \mathsf{refl}\, (t^\mathsf{E}\, \gamma^3) \\
(\mathsf{J}_{a\,t\,(x.z.p)}\, pr\, _u\, eq)^\mathsf{E}\, \gamma^3 &:\equiv
\end{aligned}
$$

$$
\mathsf{J}\left(\mathsf{J}\left(\mathsf{J}\big(\mathsf{J}\,(\lambda p^M\, p^E\, pr^M\, pr^E.pr^E)\, (t^\mathsf{E}\, \gamma^3)\right.\right.
$$
$$
\left.\left.(\mathsf{uncurry}\, p^M\, \gamma^2)\, (\mathsf{uncurry}\, p^E\, \gamma^3)\, (pr^M\, \gamma^2)\, (pr^\mathsf{E}\, \gamma^3)\big)\, eq^\mathsf{C}\, \gamma\right)\, u^\mathsf{E}\, \gamma^3\right)\, (eq^\mathsf{E}\, \gamma^3)
$$

$$
(\mathsf{J}\beta_{a\,t\,(x.z.p)}\, pr)^\mathsf{E}\, \gamma^3 \quad :\equiv
$$
$$
\mathsf{J}\big(\mathsf{J}\,(\lambda p^M\, p^E.\, \mathsf{refl})\, (t^\mathsf{E}\, \gamma^3)\, (\mathsf{uncurry}\, p^M\, \gamma^2)\, (\mathsf{uncurry}\, p^\mathsf{E}\, \gamma^3)\big)\, (pr^\mathsf{E}\, \gamma^3)
$$
$$
\begin{aligned}
((x : A) \to b)^\mathsf{E}\, \gamma^3\, f\, t &:\equiv b^\mathsf{E}\, \gamma^3\, (f\, t) \\
(t\, u)^\mathsf{E}\, \gamma^3 &:\equiv \mathsf{ap}\, (\lambda f.f\, u)\, (t^\mathsf{E}\, \gamma^3)
\end{aligned}
$$

The U^E and $(\underline{a})^\mathsf{E}$ definitions are the key points of the $-^\mathsf{E}$ operation. The definitions for the other $-^\mathsf{E}$ cases are largely determined by these.

The U^E rule yields the type of the eliminator for a type formation rule. In the natural numbers example above, the non-indexed $Nat : \mathsf{U}$ sort is interpreted as $n^E : (x{:}n) \to n^M\, x$. For indexed sorts, the indices are first processed by the $-^\mathsf{E}$ cases for inductive and non-inductive parameters, until the ultimate U return type is reached. Hence, the eliminator for a sort is always a function.

Analogously, the $-^{\mathsf{E}}$ result type for a point or path constructor is always a β-rule, i.e. a function type ending in an equality. To see why, consider the $(\underline{a})^{\mathsf{E}}$ definition. It expresses that applications of a^{E} eliminators must be equal to the corresponding t^M induction methods. Hence, for path and point constructor types, $-^{\mathsf{E}}$ works by first processing all inductive and non-inductive indices, then finally returning an equality type.

Let us also consider the $((x : a) \to B)^{\mathsf{E}}$ case for inductive parameters. Here, we make crucial use of the fact that the domain type a is small. This provides us $a^{\mathsf{E}} \gamma^3 x$, which we use to witness the $a^M \gamma^2 x$ hypothesis for B^{E}. Without the size restriction on inductive parameters (which enforces strict positivity), the $-^{\mathsf{E}}$ operation would not be possible at all, because a^{E} for a large a type would be merely an opaque relation instead of an eliminator function.

Here, we only provide abbreviated definitions for the $t\,u$, $t =_a u$, refl, J and $\mathsf{J}\beta$ cases. In the J case, we write $\mathsf{uncurry}\,p^M$ for $\lambda\gamma\,\gamma^M\,x\,x^M\,z\,z^M.\,p^M\,(\gamma, x, z)\,(\gamma^M, x^M, z^M)$ and analogously elsewhere. The full definitions can be found in the Agda formalisation. The definitions are highly constrained by the required types, and not particularly difficult to implement with the help of a proof assistant; they all involve doing successive path induction on all equalities available from induction hypotheses, with appropriately generalized induction motives.

The full $(\mathsf{J}_{a\,t\,(x.z.p)}\,pr\,_u\,eq)^{\mathsf{E}}$ definition is quite large, and, for instance, yields a very large β-rule for the higher inductive torus definition (the reader can confirm this using the Haskell implementation). Nevertheless, an implementation focused on practicality may provide smaller specialized $-^{\mathsf{E}}$ definitions for commonly used equality operations such as composition or inverses.

The circle example is a bit more interesting here:

$$(\cdot,\ S^1 : \mathsf{U},\ b : \underline{S^1},\ loop : \underline{b = b})^{\mathsf{E}}\ (\mathsf{tt},\ S^1,\ b,\ loop)\,(\mathsf{tt},\ S^{1M},\ b^M,\ loop^M)$$
$$\equiv \top \times (S^{1E} : (x : S^1) \to S^{1M}\,x) \times (b^E : S^{1E}\,b = b^M)$$
$$\times\,(loop^E : \mathsf{tr}_{(\lambda x.\mathsf{tr}_{S^{1M}}\,loop\,x\,=\,b^M)}\,b^E\,(\mathsf{tr}_{(\lambda x.\mathsf{tr}_{S^{1M}}\,loop\,(S^{1E}\,b)\,=\,x)}\,b^E\,(\mathsf{apd}\,S^{1E}\,loop))$$
$$=\,loop^M)$$

In homotopy type theory, the β-rule for $loop$ is usually just $\mathsf{apd}\,S^{1E}\,loop = loop^M$, but here all β-rules are propositional, so we need to transport with b^E to make the equation well-typed. When computing the type of $loop^E$, we start with $(\underline{b = b})^{\mathsf{E}}\,\gamma^3\,loop\,loop^M$. Next, this evaluates to $(b = b)^{\mathsf{E}}\,\gamma^3\,loop = loop^M$, and then we unfold the left hand side to get the doubly-transported expression in the result.

In Appendix A, we show how the elimination principle is computed for the two-dimensional sphere.

For another example for the translations, we consider indexed W-types which can describe a large class of inductive definitions [22]. Suppose we are in the target context $I : \mathsf{Type}_0, S : \mathsf{Type}_0, P : S \to \mathsf{Type}_0, out : S \to I, in : (s : S) \to P\,s \to I$. Then, the code for the corresponding indexed W-type is the following:

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

We pick a universe level j for elimination. The interpretations of W are the following,

omitting leading $\top$ components:

$$W^{\mathsf{C}} \equiv (w : I \to \mathsf{Type}_0) \times ((s : S) \to ((p : P\,s) \to w\,(in\,s\,p)) \to w\,(out\,s))$$

$$W^{\mathsf{M}}\,(w, sup) \equiv (w^M : (i : I) \to w\,i \to \mathsf{Type}_j)$$
$$\times\,\big((s : S)(f : (p : P\,s) \to w\,(in\,s\,p))$$
$$\to ((p : P\,s) \to w^M\,(in\,s\,p)\,(f\,p)) \to w^M\,(out\,s)\,(sup\,s\,f)\big)$$

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

5 Existence of HIITs

The target type theory supports HIITs if whenever we can derive $\Gamma \vdash \Delta$ in the source theory, the following rules are admissible.

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

We can add HIITs to the target theory by extending it with the theory of codes (making the target and the source theory the same) and adding the above rules with the additional assumption of $\Gamma \vdash \Delta$. However, this only adds HIITs with weak computation rules. To make the computation rules definitional, we would probably need a two-level target type theory with an equality having an equality reflection rule as in Voevodsky's homotopy type system [28] or Andromeda [7].

6 Formalisation and implementation

There are three additional development artifacts to the current work: a Haskell implementation, a shallow Agda formalisation and a deep Agda formalisation. All three are available from the homepage of the first author.

The Haskell implementation takes as input a file which contains a representation of a $\Gamma \vdash \Delta$ specification of a HIIT. Then, it checks the input with respect to the rules in figure 1, and outputs an Agda-checkable file which contains the results of the $-^{\mathsf{C}}$, $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ translations. It comes with examples, including the ones in this paper, indexed W-types [22], the dense completion [23, Appendix A.1.3] and several HITs from [25] including the definition of Cauchy reals. It can be checked that our implementation computes the expected elimination principles in these cases.

The shallow Agda formalisation embeds both the source and target theories shallowly into Agda: it represents types as Agda types, functions as Agda functions, and so on. We also leave the $-^{\mathsf{C}}$ operation implicit. We state each case of the $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ translations as Agda functions from all induction hypotheses to the result type of the translation, which lets us "typecheck" the translation. We have found that this style of formalisation is conveniently light, but remains detailed enough to be useful. We also generated some of the code of the Haskell implementation from this formalisation.

The deep Agda formalisation still uses a shallow embedding of the target type theory, but it uses a deep embedding of the source theory, in the style of [3]. In the formalisation we merge the three translations into a single model construction. This allows us to prove

strict preservation of definitional equalities in the substitution calculus of the source theory, in contrast to the shallow formalisation, where we cannot reason about definitional equalities of Agda terms. Due to technical challenges, this formalisation uses transport instead of J in the source theory, but this still covers a rather large class of HIIT definitions.

7   Conclusions and further work

Higher inductive-inductive types are useful in defining the well-typed syntax of type theory in an abstract way [3]. From a universal algebra point of view, they provide initial algebras for multi-sorted algebraic theories where the sorts can depend on each other. From the perspective of homotopy type theory, they provide synthetic versions of homotopy-theoretic constructions such as higher dimensional spheres or cell complexes. So far, no general scheme of HIITs have been proposed. To quote Lumsdaine and Shulman [21]:

> "The constructors of an ordinary inductive type are unrelated to each other. But in a higher inductive type each constructor must be able to refer to the previous ones; specifically, the source and target of a path-constructor generally involve the previous point-constructors. No syntactic scheme has yet been proposed for this that covers all cases of interest while remaining meaningful and consistent."

In this paper we proposed such a syntactic scheme which also includes inductive-inductive types. We tackled the problem of complex dependencies on previous type formation rules and constructors by a well-known method of describing intricate dependencies: the syntax of type theory itself. We had to limit the 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.

In this paper we only *specified* HIITs and characterised their induction principles. Proving their *existence* is left as further work. This would likely involve reducing HIITs to basic building blocks such as W-types and quotient types.

The theory of codes was defined as part of the syntax of another type theory, the target theory. An alternative way would be to define the theory of codes internally to a type theoretic metatheory in the style of [3]. However, as described in [3, Section 6], there would be a coherence problem: for the internal syntax to be useful, we need to truncate it to be a set (as the trunc constructor did for Int in the introduction). As the eliminator needs to respect the equality given by trunc, we would only be able to eliminate from the internal type theory into a set. This would preclude the definition of even the $-^\mathsf{C}$ operation, which corresponds to the standard model. A possible solution to this problem would be to add all the higher coherence laws to the syntax (e.g. the pentagon law for the composition of substitutions) and then prove that the syntax is a set. For this however, we would probably need a two-level metatheory as in [11].

When working in a metatheory with uniqueness of identity proofs (which implies that all HIITs are in essence QIITs), the theory of codes admits a category model where the interpretation of a context is the category of algebras corresponding to the context. In the future, we would like to prove that these categories have initial algebras given by terms in the theory of codes.

── **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, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham, 2018. Springer International Publishing.

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. URL: `http://dl.acm.org/citation.cfm?id=2837614`, `doi:10.1145/2837614.2837638`.

4   Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014. URL: `http://dl.acm.org/citation.cfm?id=2535838`, `doi:10.1145/2535838.2535852`.

5   Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of LICS '16*, pages 327–336. ACM, 2016. `doi:10.1145/2933575.2934514`.

6   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.

7   Andrej Bauer, Gaëtan Gilbert, Philipp Haselwarter, Matija Pretnar, and Christopher A. Stone. Design and implementation of the andromeda proof assistant. In Silvia Ghilezan and Ivetić Jelena, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016*. University of Novi Sad, 2016.

8   Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *ACM Sigplan Notices*, volume 45, pages 345–356. ACM, 2010.

9   Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012. `doi:10.1017/S0956796812000056`.

10 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. `doi:10.1145/3018610.3018620`.

11 Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-segal types. *Proc. ACM Program. Lang.*, 2(POPL):44:1–44:29, dec 2017. `doi:10.1145/3158132`.

12 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. `doi:10.1145/2854065.2854076`.

13 Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

14 Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:525–549, 2000.

15 Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. *Electronic Notes in Theoretical Computer Science*, 336:119–134, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). `doi:10.1016/j.entcs.2018.03.019`.

16   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.

17   Martin Hofmann. *Extensional concepts in intensional type theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.

18   Ambrus Kaposi. *Type theory in a type theory with quotient inductive types.* PhD thesis, University of Nottingham, 2017.

19   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. `doi:10.1145/2933575.2933586`.

20   Marc Lasson. Canonicity of weak ω-groupoid laws using parametricity theory. *Electronic Notes in Theoretical Computer Science*, 308:229–244, 2014. `doi:10.1016/j.entcs.2014.10.013`.

21   Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2017. `arXiv:1705.07088`.

22   Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

23   Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

24   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. `doi:10.1007/BFb0037116`.

25   The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

26   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. `doi:10.1145/2676726.2676983`.

27   Niels van der Weide. Higher inductive types. Master's thesis, Radboud University, Nijmegen, 2016.

28   Vladimir Voevodsky. A simple type system with two identity types. Unpublished note, 2013.

A   Elimination principle computed for the two-dimensional sphere

The two-dimensional sphere is given by the following context in the theory of codes.

$$\Gamma :\equiv \left(\cdot, \ \ S^2 : \mathsf{U}, \ \ b : \underline{S^2}, \ \ surf : \underline{\mathsf{refl}_b =_{(b=_{S^2} b)} \mathsf{refl}_b} \right)$$

The sphere-algebras are computed as follows.

$$\Gamma^{\mathsf{C}} \equiv \top \times (S^2 : \mathsf{Type}_0) \times (b : S^2) \times (surf : \mathsf{refl}_b =_{(b=_{S^2} b)} \mathsf{refl}_b)$$

Given a sphere-algebra and fixing a universe level i, the motives and methods are computed as follows.

$$\Gamma^{\mathsf{M}}\;(\mathsf{tt},\,S^2,\,b,\,surf)$$
$$\equiv\;\top\times(S^{2M}:S^2\to\mathsf{Type}_i)$$
$$\times\,(b^M:S^{2M}\,b)$$
$$\times\,(surf^M:\mathsf{tr}_{(\mathsf{tr}_{S^{2M}}\,-\,b^M=b^M)}\,surf\,\mathsf{refl}_{b^M}=\mathsf{refl}_{b^M})$$

Given a sphere-algebra and the motives and methods for this algebra, the types of the elimination principles are computed as follows.

$$\Gamma^{\mathsf{E}}\;(\mathsf{tt},\,S^2,\,b,\,surf)\,(\mathsf{tt},\,S^{2M},\,b^M,\,surf^M)$$
$$\equiv\;\top\times(S^{2E}:(x:S^2)\to S^{2M}\,x)$$
$$\times\,(b^E:S^{2E}\,b=b^M)$$
$$\times\left(surf^E:\mathsf{tr}\left(\mathsf{J}\,\mathsf{refl}\,b^E\right)\left(\mathsf{tr}\left(\mathsf{J}\,\mathsf{refl}\,b^E\right)\left(\mathsf{apd}\,(\lambda x.\mathsf{tr}\,b^E\,(\mathsf{tr}\,b^E\,(\mathsf{apd}\,S^{2E}\,x)))\,surf\right)\right)\right.$$
$$\left.=surf^M\right)$$

Note that if b^E is a definitional equality (that is, we have $S^{2E}\,b\equiv b^M$), the occurrences of b^E in the type of $surf^E$ can be replaced by refl. In this case the type of $surf^E$ becomes the expected $\mathsf{apd}\,(\mathsf{apd}\,S^{2E})\,surf=surf^M$.