


1 A Syntax for Higher Inductive-Inductive Types

2 **Ambrus Kaposi**

3 Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest,
4 Hungary

5 akaposi@inf.elte.hu

6  <https://orcid.org/0000-0001-9897-8936>

7 **András Kovács**

8 Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest,
9 Hungary

10 kovacsandras@inf.elte.hu

11  <https://orcid.org/0000-0002-6375-9781>

12 — Abstract —

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

27 **2012 ACM Subject Classification** Theory of computation \rightarrow Type theory

28 **Keywords and phrases** homotopy type theory, inductive-inductive types, higher inductive types,
29 quotient inductive types, logical relations

30 **Digital Object Identifier** 10.4230/LIPIcs.FSCD.2018.20

31 **Funding** This work was supported by the European Union, co-financed by the European So-
32 cial Fund (EFOP-3.6.3-VEKOP-16-2017-00002), USAF grant FA9550-16-1-0029, and by COST
33 Action EUTypes CA15123.

34 **Acknowledgements** The authors thank Thorsten Altenkirch, Simon Boulier, Paolo Capriotti,
35 Péter Diviánszky, Gábor Lehel and Nicolas Tabareau for discussions related to this paper and
36 the anonymous reviewers for their helpful comments and suggestions. We thank Balázs Kőműves
37 for the idea of using a Tarski universe instead of a Russell universe in the theory of codes.

38 **1** Introduction

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



© Ambrus Kaposi and András Kovács;
licensed under Creative Commons License CC-BY

3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018).

Editor: Hélène Kirchner; Article No. 20; pp. 20:1–20:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

20:2 A Syntax for Higher Inductive-Inductive Types

42 For example, the inductive type of natural numbers `Nat` is given by the constructors
 43 `zero : Nat` and `suc : Nat → Nat`. The eliminator corresponds to the usual notion of
 44 mathematical induction:

45 $\text{ElimNat} : (P : \text{Nat} \rightarrow \text{Type})(pz : P \text{ zero})(ps : (n : \text{Nat}) \rightarrow P n \rightarrow P (\text{suc } n))(n : \text{Nat}) \rightarrow P n$

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

52 $\text{ElimNat } P \text{ } pz \text{ } ps \text{ } \text{zero} \quad \equiv \text{ } pz$

53 $\text{ElimNat } P \text{ } pz \text{ } ps \text{ } (\text{suc } n) \equiv ps \text{ } n \text{ } (\text{ElimNat } P \text{ } pz \text{ } ps \text{ } n)$

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

58 Dependent families of types can be defined in a similar way, for example vectors of
 59 A -elements $\text{Vec}_A : \text{Nat} \rightarrow \text{Type}$ which are indexed by their length. Another generalisation
 60 of inductive types are mutual inductive types. However, these can be reduced to indexed
 61 families where indices classify constructors for each mutual type. Inductive-inductive types
 62 [23] are mutual definitions where this reduction does not work: here a type is defined together
 63 with a family indexed over it. An example is the following fragment of the well-typed syntax
 64 of type theory where the second sort `Ty` is indexed over the first sort `Con`, but constructors
 65 of `Con` also refer to `Ty`:

66	<code>Con</code> : Type	sort of contexts
67	<code>Ty</code> : <code>Con</code> → Type	sort of types given a context
68	<code>•</code> : <code>Con</code>	constructor for the empty context
69	<code>– ▷ –</code> : (<code>Γ</code> : <code>Con</code>) → <code>Ty</code> <code>Γ</code> → <code>Con</code>	constructor for context extension
70	<code>U</code> : (<code>Γ</code> : <code>Con</code>) → <code>Ty</code> <code>Γ</code>	constructor for a base type
71 72	<code>Π</code> : (<code>Γ</code> : <code>Con</code>)($A : \text{Ty } \Gamma$) → <code>Ty</code> (<code>Γ</code> ▷ A) → <code>Ty</code> <code>Γ</code>	constructor for dependent functions

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

76 $\text{ElimCon} : (P : \dots)(Q : \dots) \rightarrow \dots \rightarrow (\Gamma : \text{Con}) \rightarrow P \Gamma$

77 $\text{ElimTy} : (P : \dots)(Q : \dots) \rightarrow \dots \rightarrow (A : \text{Ty } \Gamma) \rightarrow Q \Gamma (\text{ElimCon } \Gamma) A$

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

81 Higher inductive types (HITs, [25, Chapter 6]) generalise inductive types in a different
 82 way: they allow constructors expressing equalities of elements of the type being defined. This
 83 enables, among others, the definition of types quotiented by a relation. For example, the
 84 type of integers `Int` can be given by a constructor `pair : Nat → Nat → Int` and an equality
 85 constructor `eq : (abcd : Nat) → a + d =Nat b + c → pair ab =Int pair cd` targeting an

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

96 Not all constructor expressions make sense. For example [25, Example 6.13.1], given an
 97 $f : (X : \text{Type}) \rightarrow X \rightarrow X$, suppose that an inductive type lval is generated by the point
 98 constructors $\mathbf{a} : \text{lval}$, $\mathbf{b} : \text{lval}$ and a path constructor $\sigma : f \text{lval } \mathbf{a} =_{\text{lval}} f \text{lval } \mathbf{b}$. The eliminator
 99 for this type should take a motive $P : \text{lval} \rightarrow \text{Type}$, two methods $p_a : P \mathbf{a}$ and $p_b : P \mathbf{b}$, and a
 100 path connecting elements of $P(f \text{lval } \mathbf{a})$ and $P(f \text{lval } \mathbf{b})$. However it is not clear what these
 101 elements should be: we only have elements $p_a : P \mathbf{a}$ and $p_b : P \mathbf{b}$, and there is no way in
 102 general to transform these to have types $P(f \text{lval } \mathbf{a})$ and $P(f \text{lval } \mathbf{b})$.

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

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

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

121
$$\text{Nat} : \mathbb{U}, \quad \text{zero} : \underline{\text{Nat}}, \quad \text{suc} : \text{Nat} \rightarrow \underline{\text{Nat}}$$

122 where Nat , zero and suc are simply variable names, and underlining denotes El (decoding)
 123 for the Tarski-style universe \mathbb{U} .

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

130 To our knowledge, this is the first proposal for a definition of HIITs. Proving the existence
 131 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 initial 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 \mathbf{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

178 and show that this corresponds to initiality.

179 The logical predicate syntactic translation was introduced by Bernardy et al. [8]. The
 180 idea that a context can be seen as a definition of an inductive type and the logical predicate
 181 translation can be used to derive the types of motives and methods was described in [3,
 182 Section 5.3]. Logical relations are used to derive the computation rules in [18, Section
 183 4.3], however only for closed QIITs. Syntactic translations in the context of the calculus of
 184 inductive constructions are discussed in [10]. Logical relations and parametricity can also
 185 be used to justify the *existence* of inductive types in a type theory with an impredicative
 186 universe [4]. In contrast, we only use logical relations to *describe* HIITs.

187 2 Target type theory

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

195 $\vdash \Gamma$ Γ is a valid target context
 196 $\Gamma \vdash t : A$ the target term t has target type A in target context Γ
 197

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

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

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

205 $(x : A) \times B$ stands for Σ types, $A \times B$ for the non-dependent version and \times associates to
 206 the left. The constructor for Σ types is denoted (t, u) with eliminators proj_1 and proj_2 . Both
 207 Π and Σ have definitional β and η rules.

208 The equality (identity) type for a type A and elements $t : A, u : A$ is denoted $t =_A u$ and
 209 comes with the constructor refl_t and eliminator J with definitional β -rule. The notation is
 210 $\text{J}_{A t P pr u eq}$ for $t : A, P : (x : A) \rightarrow t =_A x \rightarrow \text{Type}_i, pr : P t \text{refl}$ and $eq : t =_A u$. Sometimes we
 211 omit parameters in subscripts.

212 We will use the following functions defined using J in the standard way. We write
 213 $\text{tr}_{P e} t : P v$ for transport of $t : P u$ along $e : u = v$. We write $\text{ap } f e : f u = f v$ for $f : A \rightarrow B$
 214 and $e : u = v$, $\text{apd } f e : \text{tr}_{P e} (f u) = f v$ for $f : (x : A) \rightarrow B$ and $e : u = v$.

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

20:6 A Syntax for Higher Inductive-Inductive Types

(1) Contexts and variables

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

(2) Universe

$$\frac{}{\Gamma; \vdash \Delta} \quad \frac{}{\Gamma; \Delta \vdash \mathbb{U}} \quad \frac{}{\Gamma; \Delta \vdash a : \mathbb{U}} \quad \frac{}{\Gamma; \Delta \vdash \underline{a}}$$

(3) Inductive parameters

$$\frac{}{\Gamma; \Delta \vdash a : \mathbb{U}} \quad \frac{}{\Gamma; \Delta, x : \underline{a} \vdash B} \quad \frac{}{\Gamma; \Delta \vdash t : (x : a) \rightarrow B} \quad \frac{}{\Gamma; \Delta \vdash u : \underline{a}} \quad \frac{}{\Gamma; \Delta \vdash (x : a) \rightarrow B} \quad \frac{}{\Gamma; \Delta \vdash tu : B[x \mapsto u]}$$

(4) Equality

$$\frac{}{\Gamma; \Delta \vdash a : \mathbb{U}} \quad \frac{}{\Gamma; \Delta \vdash t : \underline{a}} \quad \frac{}{\Gamma; \Delta \vdash u : \underline{a}} \quad \frac{}{\Gamma; \Delta \vdash t =_a u : \mathbb{U}} \quad \frac{}{\Gamma; \Delta \vdash t : \underline{a}} \quad \frac{}{\Gamma; \Delta \vdash \text{refl} : t =_a t}$$

$$\frac{\Gamma; \Delta \vdash t : \underline{a} \quad \Gamma; \Delta, x : \underline{a}, z : t =_a x \vdash p : \mathbb{U} \quad \Gamma; \Delta \vdash pr : p[x \mapsto t, z \mapsto \text{refl}] \quad \Gamma; \Delta \vdash u : \underline{a} \quad \Gamma; \Delta \vdash eq : t =_a u}{\Gamma; \Delta \vdash J_{at} (x.z.p) pr u eq : p[x \mapsto u, z \mapsto eq]} \quad \frac{\Gamma; \Delta \vdash t : \underline{a} \quad \Gamma; \Delta, x : \underline{a}, z : t =_a x \vdash p : \mathbb{U} \quad \Gamma; \Delta \vdash pr : p[x \mapsto t, z \mapsto \text{refl}]}{\Gamma; \Delta \vdash J\beta_{at} (x.z.p) pr : (J_{at} (x.z.p) pr t \text{refl}) =_{p[x \mapsto t, z \mapsto \text{refl}]} pr}$$

(5) Non-inductive parameters

$$\frac{\Gamma \vdash A : \text{Type}_0 \quad \Gamma; \vdash \Delta \quad (\Gamma, x : A); \Delta \vdash B}{\Gamma; \Delta \vdash (x : A) \rightarrow B}$$

$$\frac{\Gamma; \Delta \vdash t : (x : A) \rightarrow B \quad \Gamma \vdash u : A}{\Gamma; \Delta \vdash tu : B[x \mapsto u]}$$

(6) Infinitary parameters

$$\frac{\Gamma \vdash A : \text{Type}_0 \quad \Gamma; \vdash \Delta \quad (\Gamma, x : A); \Delta \vdash b : \mathbb{U}}{\Gamma; \Delta \vdash (x : A) \rightarrow b : \mathbb{U}}$$

$$\frac{\Gamma; \Delta \vdash t : (x : A) \rightarrow b \quad \Gamma \vdash u : A}{\Gamma; \Delta \vdash tu : b[x \mapsto u]}$$

■ **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 Γ ; assumptions are not used or changed in parts (1)–(4).

3 Source type theory

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

219	$\Gamma \vdash \Delta$	Δ is a context in the target context Γ
220	$\Gamma; \Delta \vdash A$	A is a type in context Δ and target context Γ
221 222	$\Gamma; \Delta \vdash t : A$	t is a term of type A in context Δ and target context Γ

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 \mathbb{U} , with decoding written as an underline (usually \mathbb{E} in the literature). Type formation rules will target \mathbb{U} . With this part of the syntax we can already define contexts specifying the empty type, unit type and booleans:

234	$\cdot, \text{Empty} : \mathbb{U}$	$\cdot, \text{Unit} : \mathbb{U}, \text{tt} : \underline{\text{Unit}}$	$\cdot, \text{Bool} : \mathbb{U}, \text{true} : \underline{\text{Bool}}, \text{false} : \underline{\text{Bool}}$
-----	------------------------------------	--	--

(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 \mathbb{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, $a \rightarrow B$ can be written instead of $(x : a) \rightarrow B$.

Now we can specify the natural numbers as a context:

241	$\cdot, \text{Nat} : \mathbb{U}, \text{zero} : \underline{\text{Nat}}, \text{suc} : \text{Nat} \rightarrow \underline{\text{Nat}}$
-----	--

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

244	$\cdot, \text{Con} : \mathbb{U}, \text{Ty} : \text{Con} \rightarrow \mathbb{U}, \bullet : \underline{\text{Con}}, - \triangleright - : (\Delta : \text{Con}) \rightarrow \text{Ty } \Delta \rightarrow \underline{\text{Con}},$
245 246	$U : (\Delta : \text{Con}) \rightarrow \underline{\text{Ty } \Delta}, \Pi : (\Delta : \text{Con})(A : \text{Ty } \Delta)(B : \text{Ty } (\Delta \triangleright A)) \rightarrow \underline{\text{Ty } \Delta}$

(4) \mathbb{U} is closed under the equality type, with eliminator J and a weak (non-definitional) β -rule. Weakness is required because the syntactic translation $-^{\text{E}}$ 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:

252	$\cdot, S^1 : \mathbb{U}, \text{base} : \underline{S^1}, \text{loop} : \underline{\text{base} =_{S^1} \text{base}}$
-----	---

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 $\text{J}_{a u x.z.(t=x)} p v q : \underline{t =_a v}$. The torus is given as follows.

257 258	$\cdot, T^2 : \mathbb{U}, b : \underline{T^2}, p : \underline{b =_{T^2} b}, q : \underline{b =_{T^2} b}, t : \underline{p \bullet q =_{(b =_{T^2} b)} q \bullet p}$
------------	---

20:8 A Syntax for Higher Inductive-Inductive Types

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

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

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

269 $A : \text{Type}_0 \vdash \cdot, \text{List} : \mathbb{U}, \text{nil} : \underline{\text{List}}, \text{cons} : (x : A) \rightarrow \text{List} \rightarrow \underline{\text{List}}$
 270 $\Gamma \vdash \cdot, \text{Int} : \mathbb{U}, \text{pair} : (xy : \text{Nat}) \rightarrow \underline{\text{Int}},$
 271 $\text{eq} : (abcd : \text{Nat})(p : a+d =_{\text{Nat}} b+c) \rightarrow \underline{\text{pair } ab =_{\text{Int}} \text{pair } cd},$
 272 $\text{trunc} : (xy : \text{Int})(pq : a =_{\text{Int}} b) \rightarrow \underline{p =_{x=\text{Int } y} q}$

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

277 $A : \text{Type}_0 \vdash \cdot, \text{tr} : \mathbb{U}, \text{emb} : (x : A) \rightarrow \underline{\text{tr}}, \text{eq} : (xy : \text{tr}) \rightarrow \underline{x =_{\text{tr}} y}$

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

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

289 $A : \text{Type}_0, B : \text{Type}_0 \vdash \cdot, T : \mathbb{U}, \text{leaf} : (x : A) \rightarrow \underline{T}, \text{node} : ((x : B) \rightarrow T) \rightarrow \underline{T}$

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

293 $S : \text{Type}_0, P : S \rightarrow \text{Type}_0 \vdash \cdot, W : \mathbb{U}, \text{sup} : (s : S) \rightarrow ((p : P s) \rightarrow W) \rightarrow \underline{W}$

294 For a more complex infinitary example, see the definition of Cauchy reals in [25, Definition
 295 11.3.2]. It can be also found as an example file in our Haskell implementation.

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

298 $\cdot, \text{Ival} : \mathbb{U}, a : \underline{\text{Ival}}, b : \underline{\text{Ival}}, \sigma : ? =_{\text{Ival}}?$

299 The first argument of the function $f : (X : \text{Type}) \rightarrow X \rightarrow X$ is a target theory type, but we
 300 only have $\text{Ival} : \mathbb{U}$ in the theory of codes. `Neg` cannot be typed because the first parameter
 301 of the constructor `con` is a function from a small type to a target theory type, and no such
 302 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 $-^C$, $-^M$ and $-^E$ operations, which work as follows on the code for natural numbers.

$$\begin{aligned}
& (\cdot, \text{Nat} : \mathbf{U}, \text{zero} : \underline{\text{Nat}}, \text{suc} : \text{Nat} \rightarrow \underline{\text{Nat}})^C \\
& \equiv \top \times (n : \text{Type}_0) \times (z : n) \times (n \rightarrow n) \\
& (\cdot, \text{Nat} : \mathbf{U}, \text{zero} : \underline{\text{Nat}}, \text{suc} : \text{Nat} \rightarrow \underline{\text{Nat}})^M (\text{tt}, n, z, s) \\
& \equiv \top \times (n^M : n \rightarrow \text{Type}_i) \times (z^M : n^M z) \times ((x : n) \rightarrow n^M x \rightarrow n^M (s x)) \\
& (\cdot, \text{Nat} : \mathbf{U}, \text{zero} : \underline{\text{Nat}}, \text{suc} : \text{Nat} \rightarrow \underline{\text{Nat}})^E (\text{tt}, n, z, s) (\text{tt}, n^M, z^M, s^M) \\
& \equiv \top \times (n^E : (x : n) \rightarrow n^M x) \times (z^E : n^E z = z^M) \times ((x : n) \rightarrow n^E (s x) = n^M x (n^E x))
\end{aligned}$$

The brick red coloured result of $-^C$ gives the types of the type formation rule and constructors as an iterated Σ -type. Assuming the existence of constructors, $-^M$ returns the types of motives and methods. Assuming the existence of constructors and the corresponding motives and methods, $-^E$ 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 \mathbf{U} , \underline{a} for some neutral a , or $t =_a u$. The following table summarizes the results of the various translations in the mentioned three cases:

return type	$-^C$	$-^M$	$-^E$
\mathbf{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 $-^C$, $-^M$ and $-^E$ 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.

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

$$386 \frac{\Gamma; \Delta \vdash t : A}{\Gamma \vdash t^M : (\gamma : \Delta^C)(\gamma^M : \Delta^M \gamma) \rightarrow A^M \gamma \gamma^M (t^C \gamma)}$$

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

$$\begin{aligned}
 389 \quad .^M \gamma & \quad \equiv \top \\
 390 \quad (\Delta, x : A)^M (\gamma, t) & \quad \equiv (\gamma^M : \Delta^M \gamma) \times A^M \gamma^2 t \\
 391 \quad x^M \gamma^2 & \quad \equiv x^{\text{th}} \text{ component in } \gamma^M \\
 392 \quad \mathsf{U}^M \gamma^2 A & \quad \equiv A \rightarrow \mathsf{Type}_i \\
 393 \quad (\underline{a})^M \gamma^2 t & \quad \equiv a^M \gamma^2 t \\
 394 \quad ((x : a) \rightarrow B)^M \gamma^2 f & \quad \equiv (x : a^C \gamma)(x^M : a^M \gamma^2 x) \rightarrow B^M (\gamma, x) (\gamma^M, x^M) (f x) \\
 395 \quad (t u)^M \gamma^2 & \quad \equiv (t^M \gamma^2) (u^C \gamma) (u^M \gamma^2) \\
 396 \quad ((x : A) \rightarrow B)^M \gamma^2 f & \quad \equiv (x : A) \rightarrow B^M \gamma^2 (f x) \\
 397 \quad (t u)^M \gamma^2 & \quad \equiv t^M \gamma^2 u \\
 398 \quad (t =_a u)^M \gamma^2 e & \quad \equiv \mathsf{tr}_{(a^M \gamma^2)} e (t^M \gamma^2) = u^M \gamma^2 \\
 399 \quad (\mathsf{refl}_t)^M \gamma^2 & \quad \equiv \mathsf{refl}_{(t^M \gamma^2)} \\
 400 \quad (\mathsf{J}_{at(x.z.p)} \mathsf{pr}_u \mathsf{eq})^M \gamma^2 & \quad \equiv \mathsf{J} (\mathsf{J} (\mathsf{pr}^M \gamma^2) (\mathsf{eq}^C \gamma)) (\mathsf{eq}^M \gamma^2) \\
 401 \quad (\mathsf{J}\beta_{at(x.z.p)} \mathsf{pr})^M \gamma^2 & \quad \equiv \mathsf{refl} \\
 402 \quad ((x : A) \rightarrow b)^M \gamma^2 f & \quad \equiv (x : A) \rightarrow b^M \gamma^2 (f x) \\
 403 \quad (t u)^M \gamma^2 & \quad \equiv t^M \gamma^2 u \\
 404
 \end{aligned}$$

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

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

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

413 The predicate for the equality type $t =_a u$, for each $e : (t =_a u)^C \gamma$, i.e. $e : t^C \gamma = u^C \gamma$,
 414 says that t^M and u^M are equal. As these have different types, we have to transport over the
 415 original equality e . The witness for refl is reflexivity in the target theory. The interpretation
 416 of J is given by a double J application, which definition is sourced from [20]. Here, we use a
 417 shortened J notation; we refer to the formalisation (Section 6) for the details.

418 Again, let us consider the circle example:

$$\begin{aligned}
 419 \quad & (\cdot, S^1 : \mathsf{U}, b : S^1, \mathsf{loop} : b \equiv b)^M (\mathsf{tt}, S^1, b, \mathsf{loop}) \\
 420 \quad & \equiv \top \times (S^{1M} : S^1 \rightarrow \mathsf{Type}_i) \times (b^M : S^{1M} b) \times (\mathsf{loop}^M : \mathsf{tr}_{S^{1M}} \mathsf{loop} b^M = b^M) \\
 421
 \end{aligned}$$

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

4.3 Eliminators and computation rules

The operation $-^E$ 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 $-^M$ operation.

$$\frac{\Gamma \vdash \Delta}{\Gamma; \Delta^E : (\gamma : \Delta^C) \rightarrow \Delta^M \gamma \rightarrow \mathbf{Type}_i}$$

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

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

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

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

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

$$\begin{aligned} & .^E \gamma \gamma^M && \equiv \top \\ & (\Delta, x : A)^E (\gamma, t) (\gamma^M, t^M) && \equiv (\gamma^E : \Delta^E \gamma^2) \times A^E \gamma^3 t t^M \\ & x^E \gamma^3 && \equiv x^{\text{th}} \text{ component in } \gamma^E \\ & U^E \gamma^3 A A^M && \equiv (x : A) \rightarrow A^M x \\ & (a)^E \gamma^3 t t^M && \equiv a^E \gamma^3 t = t^M \\ & ((x : a) \rightarrow B)^E \gamma^3 f f^M && \equiv (x : a^C \gamma) \rightarrow B^E (\gamma, x) (\gamma^M, a^E \gamma^3 x) (\gamma^E, \text{refl}) \\ & && \quad (f x) (f^M x (a^E \gamma^3 x)) \\ & (t u)^E \gamma^3 && \equiv J (t^E \gamma^3 (u^C \gamma)) (u^E \gamma^3) \\ & ((x : A) \rightarrow B)^E \gamma^3 f f^M && \equiv (x : A) \rightarrow B^E \gamma^3 (f x) (f^M x) \\ & (t u)^E \gamma^3 && \equiv t^E \gamma^3 u \\ & (t =_a u)^E \gamma^3 e && \equiv \text{tr} (t^E \gamma^3) (\text{tr} (u^E \gamma^3) (\text{apd} (a^E \gamma^3) e)) \\ & (\text{refl}_t)^E \gamma^3 && \equiv J \text{refl} (t^E \gamma^3) \\ & (J_{a t (x.z.p)} pr_u eq)^E \gamma^3 && \equiv \\ & && J \left(J \left(J (\lambda p^M p^E pr^M pr^E . pr^E) (t^E \gamma^3) \right. \right. \\ & && \quad \left. \left. (\text{uncurry } p^M \gamma^2) (\text{uncurry } p^E \gamma^3) (pr^M \gamma^2) (pr^E \gamma^3) eq^C \gamma \right) u^E \gamma^3 \right) (eq^E \gamma^3) \\ & (J\beta_{a t (x.z.p)} pr)^E \gamma^3 && \equiv \\ & && J (J (\lambda p^M p^E . \text{refl}) (t^E \gamma^3) (\text{uncurry } p^M \gamma^2) (\text{uncurry } p^E \gamma^3)) (pr^E \gamma^3) \\ & ((x : A) \rightarrow b)^E \gamma^3 f t && \equiv b^E \gamma^3 (f t) \\ & (t u)^E \gamma^3 && \equiv \text{ap} (\lambda f . f u) (t^E \gamma^3) \end{aligned}$$

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

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

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

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

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

483 The full $(J_{at(x.z.p)} pr_u eq)^E$ definition is quite large, and, for instance, yields a very large
 484 β -rule for the higher inductive torus definition (the reader can confirm this using the Haskell
 485 implementation). Nevertheless, an implementation focused on practicality may provide
 486 smaller specialized $-^E$ definitions for commonly used equality operations such as composition
 487 or inverses.

488 The circle example is a bit more interesting here:

$$\begin{aligned}
 & (\cdot, S^1 : U, b : \underline{S^1}, \text{loop} : \underline{b = b})^E (\text{tt}, S^1, b, \text{loop}) (\text{tt}, S^{1M}, b^M, \text{loop}^M) \\
 & \equiv \top \times (S^{1E} : (x : S^1) \rightarrow S^{1M} x) \times (b^E : S^{1E} b = b^M) \\
 & \quad \times (\text{loop}^E : \text{tr}_{(\lambda x. \text{tr}_{S^{1M}} \text{loop } x = b^M)} b^E (\text{tr}_{(\lambda x. \text{tr}_{S^{1M}} \text{loop } (S^{1E} b) = x)} b^E (\text{apd } S^{1E} \text{ loop}))) \\
 & \quad = \text{loop}^M
 \end{aligned}$$

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

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

501 For another example for the translations, we consider indexed W-types which can describe
 502 a large class of inductive definitions [22]. Suppose we are in the target context $I : \text{Type}_0, S :$
 503 $\text{Type}_0, P : S \rightarrow \text{Type}_0, \text{out} : S \rightarrow I, \text{in} : (s : S) \rightarrow P s \rightarrow I$. Then, the code for the corresponding
 504 indexed W-type is the following:

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

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

507 omitting leading \top components:

$$\begin{aligned}
508 \quad W^C &\equiv (w : I \rightarrow \text{Type}_0) \times ((s : S) \rightarrow ((p : P s) \rightarrow w (in\ s\ p)) \rightarrow w (out\ s)) \\
509 \quad W^M (w, sup) &\equiv (w^M : (i : I) \rightarrow w\ i \rightarrow \text{Type}_j) \\
510 &\quad \times ((s : S)(f : (p : P s) \rightarrow w (in\ s\ p)) \\
511 &\quad \rightarrow ((p : P s) \rightarrow w^M (in\ s\ p) (f\ p)) \rightarrow w^M (out\ s) (sup\ s\ f)) \\
512 \quad W^E (w, sup) (w^M, sup^M) &\equiv \\
513 &\quad (w^E : (i : I)(x : w\ i) \rightarrow w^M\ i\ x) \\
514 &\quad \times ((s : S)(f : (p : P s) \rightarrow w (in\ s\ p)) \\
515 &\quad \rightarrow w^E (out\ s) (sup\ s\ f) = sup^M\ s\ f (\lambda p. w^E (in\ s\ p) (f\ p)))
\end{aligned}$$

517 5 Existence of HIITs

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

$$\begin{array}{c}
\hline
\Gamma \vdash \text{con}_\Delta : \Delta^C \\
\hline
\end{array}
\qquad
\begin{array}{c}
\Gamma \vdash m : \Delta^M \text{con}_\Delta \\
\hline
\Gamma \vdash \text{elim}_\Delta m : \Delta^E \text{con}_\Delta m \\
\hline
\end{array}$$

521 We can add HIITs to the target theory by extending it with the theory of codes (making
522 the target and the source theory the same) and adding the above rules with the additional
523 assumption of $\Gamma \vdash \Delta$. However, this only adds HIITs with weak computation rules. To make
524 the computation rules definitional, we would probably need a two-level target type theory
525 with an equality having an equality reflection rule as in Voevodsky’s homotopy type system
526 [28] or Andromeda [7].

527 6 Formalisation and implementation

528 There are three additional development artifacts to the current work: a Haskell implementa-
529 tion, a shallow Agda formalisation and a deep Agda formalisation. All three are available
530 from the homepage of the first author.

531 The Haskell implementation takes as input a file which contains a representation of a
532 $\Gamma \vdash \Delta$ specification of a HIIT. Then, it checks the input with respect to the rules in figure
533 1, and outputs an Agda-checkable file which contains the results of the $-^C$, $-^M$ and $-^E$
534 translations. It comes with examples, including the ones in this paper, indexed W-types
535 [22], the dense completion [23, Appendix A.1.3] and several HITs from [25] including the
536 definition of Cauchy reals. It can be checked that our implementation computes the expected
537 elimination principles in these cases.

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

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

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

552 **7** Conclusions and further work

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

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

564 In this paper we proposed such a syntactic scheme which also includes inductive-inductive
 565 types. We tackled the problem of complex dependencies on previous type formation rules
 566 and constructors by a well-known method of describing intricate dependencies: the syntax of
 567 type theory itself. We had to limit the type formers to only allow strictly positive definitions,
 568 but these restrictions are the only things that a type theorist has to learn to understand our
 569 codes. Our encoding is also *direct* in the sense that the types of constructors and eliminators
 570 are exactly as required and not merely up to isomorphisms.

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

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

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

590 **A** Elimination principle computed for the two-dimensional sphere

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

$$592 \quad \Gamma := \left(\cdot, S^2 : \mathbb{U}, b : \underline{S^2}, surf : \underline{\text{refl}_b =_{(b=S^2 b)} \text{refl}_b} \right)$$

593 The sphere-algebras are computed as follows.

$$594 \quad \Gamma^C \equiv \top \times (S^2 : \text{Type}_0) \times (b : S^2) \times (surf : \text{refl}_b =_{(b=S^2 b)} \text{refl}_b)$$

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

$$598 \quad \Gamma^M (\text{tt}, S^2, b, surf)$$

$$599 \quad \equiv \top \times (S^{2M} : S^2 \rightarrow \text{Type}_i)$$

$$600 \quad \times (b^M : S^{2M} b)$$

$$601 \quad \times (surf^M : \text{tr}_{(\text{tr}_{S^{2M}} - b^M = b^M)} surf \text{refl}_{b^M} = \text{refl}_{b^M})$$

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

$$605 \quad \Gamma^E (\text{tt}, S^2, b, surf) (\text{tt}, S^{2M}, b^M, surf^M)$$

$$606 \quad \equiv \top \times (S^{2E} : (x : S^2) \rightarrow S^{2M} x)$$

$$607 \quad \times (b^E : S^{2E} b = b^M)$$

$$608 \quad \times \left(surf^E : \text{tr} \left(\text{J refl } b^E \right) \left(\text{tr} \left(\text{J refl } b^E \right) \left(\text{apd} (\lambda x. \text{tr } b^E (\text{tr } b^E (\text{apd } S^{2E} x))) \right) surf \right) \right)$$

$$609 \quad \quad \quad = surf^M$$

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

614 — **References** —

- 615 **1** Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers — constructing strictly
616 positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Se-
617 mantics: Selected Topics.
- 618 **2** Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nord-
619 vall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, ed-
620 itors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham,
621 2018. Springer International Publishing.
- 622 **3** Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient in-
623 ductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd*
624 *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*
625 *POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016.
626 doi:10.1145/2837614.2837638.
- 627 **4** Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of
628 dependent type theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st*
629 *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*
630 *POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014.
631 doi:10.1145/2535838.2535852.

- 632 **5** Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and
633 Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors,
634 *Proceedings of LICS '16*, pages 327–336. ACM, 2016. doi:10.1145/2933575.2934514.
- 635 **6** Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in
636 programming. *Journal of Universal Computer Science*, 23(1):63–88, jan 2017.
- 637 **7** Andrej Bauer, Gaëtan Gilbert, Philipp Haselwarter, Matija Pretnar, and Christopher A.
638 Stone. Design and implementation of the andromeda proof assistant. In Silvia Ghilezan and
639 Ivetić Jelena, editors, *22nd International Conference on Types for Proofs and Programs,*
640 *TYPES 2016*. University of Novi Sad, 2016.
- 641 **8** Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent
642 types. In *ACM Sigplan Notices*, volume 45, pages 345–356. ACM, 2010.
- 643 **9** Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — paramet-
644 ricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012.
645 doi:10.1017/S0956796812000056.
- 646 **10** Simon Boulrier, Pierre-Marie Pédro, and Nicolas Tabareau. The next 700 syntactical models
647 of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs*
648 *and Proofs*, CPP 2017, pages 182–194, New York, NY, USA, 2017. ACM. doi:10.1145/
649 3018610.3018620.
- 650 **11** Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-segal
651 types. *Proc. ACM Program. Lang.*, 2(POPL):44:1–44:29, December 2017. doi:10.1145/
652 3158132.
- 653 **12** Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In
654 *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP
655 2016, pages 122–129, New York, NY, USA, 2016. ACM. doi:10.1145/2854065.2854076.
- 656 **13** Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.
- 657 **14** Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type
658 theory. *Journal of Symbolic Logic*, 65:525–549, 2000.
- 659 **15** Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model.
660 *Electronic Notes in Theoretical Computer Science*, 336:119 – 134, 2018. The Thirty-third
661 Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).
662 doi:10.1016/j.entcs.2018.03.019.
- 663 **16** Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions.
664 In *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer*
665 *Science*, pages 129–146. Springer, 1999.
- 666 **17** Martin Hofmann. *Extensional concepts in intensional type theory*. Thesis. University of
667 Edinburgh, Department of Computer Science, 1995.
- 668 **18** Ambrus Kaposi. *Type theory in a type theory with quotient inductive types*. PhD thesis,
669 University of Nottingham, 2017.
- 670 **19** Nicolai Kraus. Constructions with non-recursive higher inductive types. In *Proceedings of*
671 *the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages
672 595–604, New York, NY, USA, 2016. ACM. doi:10.1145/2933575.2933586.
- 673 **20** Marc Lasson. Canonicity of weak ω -groupoid laws using parametricity theory. *Electronic*
674 *Notes in Theoretical Computer Science*, 308:229 – 244, 2014. doi:10.1016/j.entcs.2014.
675 10.013.
- 676 **21** Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2017.
677 arXiv:1705.07088.
- 678 **22** Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Sym-*
679 *posium in Logic in Computer Science (LICS 2009)*, 2009.
- 680 **23** Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University,
681 2013.

- 682 **24** Christine Paulin-Mohring. Inductive definitions in the system Coq — rules and properties.
683 In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*,
684 *International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht,*
685 *The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer*
686 *Science*, pages 328–345. Springer, 1993. doi:10.1007/BFb0037116.
- 687 **25** The Univalent Foundations Program. Homotopy type theory: Univalent foundations of
688 mathematics. Technical report, Institute for Advanced Study, 2013.
- 689 **26** Kristina Sojakova. Higher inductive types as homotopy-initial algebras. In *Proceedings*
690 *of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*
691 *Languages*, POPL '15, pages 31–42, New York, NY, USA, 2015. ACM. doi:10.1145/
692 2676726.2676983.
- 693 **27** Niels van der Weide. Higher inductive types. Master's thesis, Radboud University, Nijme-
694 gen, 2016.
- 695 **28** Vladimir Voevodsky. A simple type system with two identity types. Unpublished note,
696 2013.