Journal of Universal Computer Science, vol. 23, no. 1 (2017), 63-88
submitted: 4/8/16, accepted: 3/12/16, appeared: 28/1/17 © J.UCS

Higher Inductive Types in Programming

Henning Basold
(Radboud University and CWI Amsterdam, The Netherlands
h.basold@cs.ru.nl)

Herman Geuvers
(Radboud University and Technical University Eindhoven, The Netherlands
herman@cs.ru.nl)

Niels van der Weide
(Radboud University, The Netherlands
nweide@cs.ru.nl)

Abstract: We propose general rules for higher inductive types with non-dependent and dependent elimination rules. These can be used to give a formal treatment of *data types with laws* as has been discussed by David Turner in his earliest papers on Miranda [Turner(1985)]. The non-dependent elimination scheme is particularly useful for defining functions by recursion and pattern matching, while the dependent elimination scheme gives an induction proof principle. We have rules for non-recursive higher inductive types, like the integers, but also for recursive higher inductive types like the truncation. In the present paper we only allow path constructors (so there are no higher path constructors), which is sufficient for treating various interesting examples from functional programming, as we will briefly show in the paper: arithmetic modulo, integers and finite sets.
Key Words: Functional Programming, Homotopy Type Theory, Higher Inductive Types
Category: D.3.1, F.4.m

1 Introduction

Already in the early days of programming it has been observed that type systems can help to ensure certain basic correctness properties of programs. For example, type systems can prevent the confusion of an integer value for a string value inside a memory cell. Much research and literature has since been devoted to type systems that allow more and more properties of programs to be checked, while retaining decidability of type checking, see [Pierce(2002), Pierce(2004)].

The very idea of using types to ensure some basic correctness properties stems from the realm of logic, namely from the monumental project of Russell and Whitehead [Whitehead and Russell(1912)] to find a logical foundation of mathematics. Since then, type systems had not been very successful in logic until Martin-Löf proposed a type system, now called Martin-Löf type theory (MLTT), that gives a computational reading to intuitionistic higher-order logic

[Martin-Löf(1975), Martin-Löf(1982), Nordström et al.(1990)] based on Russell's theory of types [Russell(1996)]. This turned type systems from tools to merely ensure correctness properties into first-class logics.

The main idea underlying MLTT is that terms (i.e., programs) can be used inside types, we say that MLTT has *dependent* types. For example, given two terms s, t, one can form a type $s = t$. Its inhabitants, that is terms of type $s = t$, should be thought of as proofs for the identity of s and t. It was then also realized that dependent types can be used to give even stronger correctness specifications of programs. For instance, suppose we can form for a type A and natural number n a type $\mathrm{Vec}\,A\,n$, the elements of which are lists over A of length n. This type allows us, for instance, to write a safe function head: $\mathrm{Vec}\,A\,(n+1) \to A$ that returns the first element of a given list. Hence, dependent types allow us to establish statically verifiable invariants based on runtime data.

Invariants as the one described above are very useful, but we often want to express more sophisticated invariants through types. An example is the type $\mathrm{Fin}(A)$ of finite subsets of a given type A. Finite sets are generated by the empty set, the singleton sets and the union of two sets together with a bunch of equations for these operations. For instance, the empty set should be neutral with respect to the union: $\emptyset \cup X = X = X \cup \emptyset$. In many programming languages this would be implemented by using lists over A as underlying type and exposing $\mathrm{Fin}(A)$ through the three mentioned operations as interface. The implementation of these operations then needs to maintain some invariants of the underlying lists, such that the desired equations hold. If these equations are used to prove correctness properties of programs, then the programmer needs to prove that the interface indeed preserves the invariants. This is a laborious task and is thus very often not carried out. So we may ask to what extent data types can be specified by an interface and invariants.

A possible extension of type systems to deal with this are quotient types. These are available in a few functional programming languages, for example Miranda [Turner(1985)], where they are called *algebraic data types with associated laws* [Thompson(1986), Thompson(1990)]. On the other hand, in the proof assistant NuPRL quotient types are implemented using squash types and non-determinism [Nogin(2002)]. In dependent types they have been introduced in a limited form in [Barthe and Geuvers(1995)], where they are called *congruence types*, and in [Hofmann(1995)]. Quotient types are fairly easy to use but have a major drawback: quotients of types whose elements are infinite, like general function spaces, often require some form of the axiom of choice, see for example [Chapman et al.(2015)]. Moreover, quotient types detach the equational specification of a data type from its interface, thus making their specification harder to read. This is because the type and its equality are defined separately.

Both problems can be fixed through by using of higher inductive types

[Lumsdaine and Shulman(2012), Lumsdaine(2011), Sojakova(2015)], and some examples are given in [The Univalent Foundations Program(2013)]. A limited form of higher inductive types has been proposed in [Altenkirch et al.(2016)] where they are defined using dialgebras. These are inductively constructed types, but unlike inductive types, one can also specify propositional equalities in their definition. So, there are not just term constructors, but also path constructors, and the elimination rule also depends on the constructors for the path.

In this paper, we propose a general scheme for higher inductive types (HITs) with non-dependent and dependent elimination rules and associated computation rules. We demonstrate our scheme through the use of HITs as replacement for quotient types in programming by studying some illustrative examples. We begin with arithmetic on integers modulo a fixed number. This example serves as an introduction to the concept of higher inductive types, and the structures and principles that are derived from their specification. Next, we give several descriptions of the integers and study their differences. Especially interesting here is that the elements of two HITs can be the same but the equality of one type can be decidable whereas that of the other is not. The last example we give are finite subsets of a given type. We show how set comprehension for finite sets can be defined. All the examples are accompanied with proofs of some basic facts that illustrate the proof principles coming with higher inductive types.

The rest of the paper is structured as follows. We first give in Section 2 a brief introduction to Martin-Löf type theory and the language of homotopy type theory, as far as it is necessary. Next, we introduce in Section 3 the syntax for the higher inductive types we will use throughout the paper. This is based on the Master's thesis of the third author [van der Weide(2016)], which also discusses the semantics of HITs that are not recursive in the equality constructors. In the following sections we study the mentioned examples of modulo arithmetic (Section 4), integers (Section 5) and finite sets (Section 6). We close with some final remarks and possibilities for future work in Section 7.

The results have been formalized in Coq using the homotopy type theory library in [Bauer et al.(2016)].

2 Martin-Löf Type Theory and Homotopy Type Theory

In this section, we introduce the variant of Martin-Löf type theory (MLTT) [Nordström et al.(1990)] that we are going to use throughout the paper, and we introduce homotopy type theory [The Univalent Foundations Program(2013)]. This type theory has as type constructors dependent function spaces (also known as Π-types), dependent binary products (aka Σ-types), binary sum types (coproducts) and identity types. Later, in Section 3, we will extend the type theory with higher inductive types, which will give us some base types like natural numbers.

Next, we will restate some well-known facts about MLTT and the identity types in particular. The properties of identity types lead us naturally towards the terminology of homotopy theory, which we discuss at the end of the section.

2.1 Martin-Löf Type Theory

We already argued in the introduction for the usefulness of dependent type theories, so let us now come to the technical details of how to realize such a theory. The most difficult part of defining such a theory is the fact that contexts, types, terms and computation rules have to be given simultaneously, as these rules use each other. Thus the following rules should be taken as simultaneous inductive definition of a calculus.

We begin by introducing a notion of context. The purpose of contexts is to capture the term variables and their types that can be used in a type, which makes the type theory dependent, or a term. These can be formed inductively by the following two rules.

$$\frac{}{\vdash \cdot \quad \textsc{Ctx}} \qquad \frac{\vdash \Gamma \quad \textsc{Ctx} \qquad \Gamma \vdash A : \textsc{Type}}{\vdash \Gamma, x : A \quad \textsc{Ctx}}$$

Note that in the second rule the type A may use variables in Γ, thus the order of variables in a context is important. We adopt the convention to leave out the empty context $\cdot$ on the left of a turnstile, whenever we give judgments for term or type formations.

The next step is to introduce judgments for kinds, types and terms. Here, the judgment $\Gamma \vdash A : \textsc{Type}$ says that A is a well-formed type in the context Γ, while $\Gamma \vdash t : A$ denotes that t is a well-formed term of type A in context Γ. For kinds we only have the following judgment.

$$\frac{\vdash \Gamma \quad \textsc{Ctx}}{\Gamma \vdash \textsc{Type} : \textsc{Kind}}$$

To ease readability, we adopt the following convention.

Notation 1. If we are given a type B with $\Gamma, x : A \vdash B : \textsc{Type}$ and a term $\Gamma \vdash t : A$, we denote by $B[t]$ the type in which t has been substituted for x. In particular, we indicate that B has x as free variable by writing $B[x]$.

The type formation rules for dependent function spaces, dependent binary products and sum types, and the corresponding term formation rules are given as follows. To avoid duplication of rules, we use $\square$ to denote either $\textsc{Type}$ or

KIND. Thus, we write $\Gamma \vdash M : \Box$ whenever M is a type or the universe TYPE.

$$\frac{\Gamma, x : A \vdash M : \Box}{\Gamma \vdash (x : A) \to M : \Box} \qquad \frac{\Gamma, x : A \vdash B : \text{TYPE}}{\Gamma \vdash (x : A) \times B : \text{TYPE}} \qquad \frac{\Gamma \vdash A, B : \text{TYPE}}{\Gamma \vdash A + B : \text{TYPE}}$$

$$\frac{\Gamma, x : A \vdash M : \Box \qquad \Gamma, x : A \vdash t : M}{\Gamma \vdash \lambda x.t : (x : A) \to M}$$

$$\frac{\Gamma, x : A \vdash M : \Box \qquad \Gamma \vdash t : (x : A) \to M \qquad \Gamma \vdash s : A}{\Gamma \vdash t\, s : M[s]}$$

$$\frac{\Gamma \vdash t : (x : A) \times B[x]}{\Gamma \vdash \pi_1\, t : A} \qquad \frac{\Gamma \vdash t : (x : A) \times B[x]}{\Gamma \vdash \pi_2\, t : B[\pi_1\, t]} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B[t]}{\Gamma \vdash (t, s) : (x : A) \times B[x]}$$

$$\frac{j \in \{1, 2\} \qquad \Gamma \vdash t : A_j}{\Gamma \vdash \text{in}_j\, t : A_1 + A_2}$$

$$\frac{\Gamma, z : A + B \vdash M : \Box \qquad \Gamma, x : A \vdash t : M[\text{in}_1\, x] \qquad \Gamma, y : B \vdash s : M[\text{in}_2\, y]}{\Gamma \vdash \{\text{in}_1\, x \mapsto t\ ;\ \text{in}_2\, y \mapsto s\} : (z : A + B) \to M}$$

If $\Gamma \vdash A, B$: TYPE, then we write $A \to B$ and $A \times B$ instead of $(x : A) \to B$ and $(x : A) \times B$, respectively.

Note that we can obtain two kinds of function spaces: $A \to B$ for a type B and $A \to$ TYPE. The latter models families of types indexed by the type A. Also note that the elimination rule for the sum type gives us what is called *large elimination*, in the sense that we can eliminate a sum type to produce a new type by case distinction. For instance, later we can define the unit type $\mathbf{1}$ as an inductive type and then a type family

$$X = \{\text{in}_1\, x \mapsto A\ ;\ \text{in}_2\, y \mapsto B\} : \mathbf{1} + \mathbf{1} \to \text{TYPE},$$

such that $X\, t$ reduces to either A or B, depending on t.

Next, identity types and their introduction and elimination terms are given by the following rules.

$$\frac{\Gamma \vdash A : \text{TYPE} \qquad \Gamma \vdash s, t : A}{\Gamma \vdash s = t : \text{TYPE}} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \text{refl}\ t : t = t}$$

$$\frac{\Gamma, x : A, y : A, p : x = y \vdash Y : \text{TYPE} \qquad \Gamma \vdash t : (x : A) \to Y[x, x, \text{refl}\ x]}{\Gamma \vdash J_{x,y,p}(t) : (x\, y : A) \to (p : x = y) \to Y[x, y, p]}$$

Higher inductive types will allow us to add more constructors, besides refl, to identity types. This will, surprisingly so, not affect the elimination principle given by J. The J-rule does not imply uniqueness of identity proofs, so it is also valid if there are other equality proofs. We discuss as part of the introduction to homotopy type theory.

To be able to evaluate computations in MLTT, we introduce a rewriting relation $\longrightarrow$ on terms and types [Nordström et al.(1990)]. This rewriting relation is given on terms as the compatible closure of the following clauses.

$$\begin{aligned}(\lambda x.t)s &\longrightarrow t[s/x]\\ \pi_k\ (t_1, t_2) &\longrightarrow t_k\\ \{\mathrm{in}_1\ x_1 \mapsto t_1\ ;\ \mathrm{in}_2\ x_2 \mapsto t_2\}\ (\mathrm{in}_k\ s) &\longrightarrow t_k[s/x_k]\\ J_{x,y,p}(t)\ s\ s\ (\mathrm{refl}\ \ s) &\longrightarrow t\ s\end{aligned}$$

On types, the reduction relation is obtained as the compatible closure of

$$\frac{s \longrightarrow t}{Y\ s \longrightarrow Y\ t}$$

Let us denote the relation for reductions in either direction by $\longleftrightarrow := \longleftarrow \cup \longrightarrow$. That is to say, we have $s \longleftrightarrow t$ if either $s \longrightarrow t$ or $t \longrightarrow s$. Moreover, we obtain *definitional equivalence*, denoted by $\equiv$, as the equivalence closure of the rewriting relation. Since definitionally equal terms are considered to carry the same information, we use the following *conversion rule* that allows us to mix rewriting steps in types with type checking.

$$\frac{\Gamma \vdash X, Y : \mathrm{Type} \qquad \Gamma \vdash u : X \qquad X \longleftrightarrow Y}{\Gamma \vdash u : Y} \tag{1}$$

By repeatedly applying this rule, we can also replace $X \longleftrightarrow Y$ by $X \equiv Y$ in it.

Let us now establish some facts about identity types, which will prove very useful later and are also relevant to the discussion of homotopy type theory. First of all, we can prove that the identity is symmetric and transitive, thus an equivalence relation. In type theoretical terms we establish that for each type A there are terms symm_A and trans_A, as indicated below. We also say that the corresponding types are *inhabited*.

$$\begin{aligned}&\mathrm{symm}_A : (x\ y : A) \to (x = y) \to (y = x)\\ &\mathrm{trans}_A : (x\ y\ z : A) \to (x = y) \to (y = z) \to (x = z)\end{aligned}$$

Proof. To demonstrate a typical use of the J-rule, let us prove transitivity by giving the corresponding term trans_A. We put

$$\begin{aligned}&Y[x, y, p] := (z : A) \to (y = z) \to (x = z)\\ &t := \lambda x\ z\ q.q,\end{aligned}$$

so $t : (x : A) \to (z : A) \to (x = z) \to (x = z)$, hence $t : (x : A) \to Y[x, x, \mathrm{refl}\ \ x]$. These definitions give us then that

$$J_{x,y,p}(t) : (x\ y : A) \to (x = y) \to (z : A) \to (y = z) \to (x = z),$$

thus

$$\mathrm{trans}_A := \lambda x\ y\ z\ q.J_{x,y,p}(t)\ x\ y\ q\ z$$

is of the correct type. $\square$

In a similar spirit, one can use the J-rule to also prove the following facts about identity types.

Proposition 2. *Let* $X \vdash \textsc{Type} :$ *and* $x : X \vdash Y[x] : \textsc{Type}$ *be types. There are terms of the following types.*

$$\vdash \mathrm{ap} : (f : X \to Y) \to (x\, y : X) \to x = y \to f\, x = f\, y$$
$$\vdash \mathrm{transport} : (x\, y : X) \to x = y \to Y[x] \to Y[y]$$

The latter we abbreviate to

$$p_* := \mathrm{transport}\ x\, y\, p.$$

This allows us to define a term

$$\vdash \mathrm{apd} : (f : (x : X) \to Y[x]) \to (x\, y : X) \to (p : x = y) \to p_*(f\, x) = f\, y.$$

We also can derive the following definitional equivalences for these terms

$$\begin{aligned} \mathrm{ap}\ f\, t\, t\, (\mathrm{refl}\ t) &\equiv \mathrm{refl}\, (f\, x) \\ \mathrm{transport}\ t\, t\, (\mathrm{refl}\ t)\, s &\equiv \mathrm{refl}\ s \\ \mathrm{apd}\ f\, t\, t\, (\mathrm{refl}\ t) &\equiv \mathrm{refl}\, (f\, x) \end{aligned}$$

Note that the names “ap” and “apd” stand for “apply” and “dependent apply”, respectively. Also, note that transport is Leibniz’ law.

Since the kind of equality that occurs in the type of apd appears frequently in the following, we use the more symmetric notation

$$s =_p^Y t \quad := \quad (p_*\, s) = t,$$

where $x : X \vdash Y[x]$ is a type, $x, y : X$, $s : Y[x]$, $t : Y[y]$ and $p : x = y$, so this denotes an equality in the type $Y[y]$.

Using this notation, apd has the following type.

$$\vdash \mathrm{apd} : (f : (x : X) \to Y[x]) \to (x\, y : X) \to (p : x = y) \to f\, x =_p^Y f\, y$$

We abbreviate ap $f\, x\, y\, p$ by $\mathrm{ap}(f, p)$ and apd $f\, x\, y\, p$ by $\mathrm{apd}(f, p)$.

2.2 Homotopy Type Theory

We have discussed several types now, and most of these have a clear meaning. For example, product types should be seen as the type of pairs. For the identity type, however, it is more complicated. An inhabitant $p : a = b$ is supposed to be a proof that a and b are equal.

In homotopy type theory types T are seen as spaces X, inhabitants $x : X$ are seen as points of X, and inhabitants $p : a = b$ are seen as paths between the points a and b. The path refl a is interpreted as the constant path. For example, the type $\mathbb{N}$ is the space with points x_n for every natural number n, and the only paths are constant paths. But we could also look at types in which there are more paths from a to b. For example, we could look at the interval which has two points 0 and 1 and a path seg between 0 and 1. Now there are two paths from 0 to 0, namely the constant path, but we can also first go from 0 to 1 via seg and then go back.

This seems rather boring now, because the most common types in type theory just have a trivial interpretation. They just consist of points, and we cannot find any non-constant path. However, one of the important features of homotopy type theory is higher inductive types which allow us to add paths to types. Even though new paths are added, the J-rule will still hold. For normal spaces this is not strange: the J-rule says how every constant path is mapped which is sufficient to define a map.

There are also two other features of homotopy type theory, but they do not play a major role in this paper. These are function extensionality and univalence. Univalence roughly says that isomorphic types are equal, and using this axiom one can prove function extensionality.

Before studying higher inductive types in Section 3, we first need to introduce some preliminary facts. For given $s, t, u : A$, $p : s = t$ and $q : t = u$ we denote the corresponding symmetry and transitivity proofs by

$$p^{-1} := \mathrm{symm}_A\ s\ t\ p$$
$$p \bullet q := \mathrm{trans}_A\ s\ t\ u\ p\ q.$$

These can be interpreted as operations on paths. The path p^{-1} is made by reversing p, and the path $p \bullet q$ is the path which starts by walking along p and then q. Again we abbreviate $\mathrm{apd}(f, x, y, p)$ by $\mathrm{apd}(f, p)$.

It is often required in homotopy type theory to compute the map p_* more concretely, and we shall do so as well. For a proof, we refer the reader to Theorem 2.11.3 in [The Univalent Foundations Program(2013)]. It is expressed as a composition of paths which is easier to determine in concrete situations.

Proposition 3. *Let A and B be types and $f, g : A \to B$ be function terms. Furthermore, suppose that we have inhabitants $a, a' : A$ and paths $p : a = a'$ and $q : f\ a = g\ a$. There is a path of type*

$$p_*(q) = \mathrm{ap}(f, p)^{-1} \bullet q \bullet \mathrm{ap}(g, p),$$

where p_* transports along $Y := f\ x = g\ x$.

3 Higher Inductive Types

Regular inductive types are usually specified by their constructors, which then give rise to canonical elimination principles, in the form of recursion or induction, and the corresponding computation principles. A higher inductive type (HIT) can additionally be equipped with path constructors for that type. The examples discussed in this paper just require paths between points, so our syntax will be restricted to this case and will not allow constructors for paths between paths. A semantical justification of this syntax has been proposed in [van der Weide(2016)]. There, the semantics is given for non-recursive higher inductive types, that is, HITs in which the path constructors do not quantify over the HIT that is being defined.

As already mentioned, a higher inductive type T can have regular data constructors and path constructors. Data constructors can take as argument a polynomial over T, which is the first notion we introduce in this section. Afterwards, we introduce a special kind of terms, called constructor terms, that will be allowed in the path constructors. These two definitions will then allow us to give (dependent) elimination principles and well-behaved computation rules for HITs.

The syntax of higher inductive types consists of two parts. First, we have the standard inductive type with a number of point constructors. On top of that, higher inductive type allow the specification of paths between elements of that type. Thus, we need to devise a syntax for adding path constructors between two elements of the type at hand.

We begin by introducing polynomial type constructors that allow us to give well-behaved constructor argument types. They ensure that a (higher) inductive type given in our syntax is strictly positive. To ease readability in the following definitions, we use the following notations for terms $t : A \to C$ and $s : B \to D$.

$$\begin{aligned}
\mathrm{id}_A &:= \lambda x.x && : A \to A \\
t \times s &:= \lambda x.(t\ (\pi_1\ x), s\ (\pi_2\ x)) && : A \times B \to C \times D \\
t + s &:= \{\mathrm{in}_1\ x \mapsto \mathrm{in}_1\ (t\ x)\ ;\ \mathrm{in}_2\ y \mapsto \mathrm{in}_2\ (s\ y)\} && : A + B \to C + D
\end{aligned}$$

Definition 4. Let X be a variable. We say that F is a *polynomial (type constructor)* if it is given by the following grammar.

$$F, G ::= A : \textsc{Type} \mid X \mid F \times G \mid F + G$$

For a type B, we denote by $F[B]$ the type that is obtained by substituting B for the variable X and interpreting $\times$ and $+$ as type constructors. Let H be a polynomial and $f : B \to C$ be a term. We define a term $H[f] : H[B] \to H[C]$, the action of H on f, by induction in H as follows.

$$\begin{aligned}
A[f] &:= \mathrm{id}_A & \qquad (F \times G)[f] &:= F[f] \times G[f] \\
X[f] &:= f & \qquad (F + G)[f] &:= F[f] + G[f]
\end{aligned} \qquad \blacktriangleleft$$

Remark. The notion of polynomial could be generalized to that of containers [Abbott et al.(2005)] or in the sense of [Gambino and Kock(2013)]. However, we stick to the above simple definition to make the development, especially the lifting to type families, more accessible.

To give the dependent elimination principle for higher inductive types, we need to be able to lift polynomials to type families (predicates) and maps between them. This is provided by the following definition.

Definition 5. Suppose F is a polynomial type constructor. We define a lifting of F to type families as follows. Let $\vdash U : B \to \textsc{Type}$ be a type family, then we can define $\vdash \overline{F}(U) : F[B] \to \textsc{Type}$ by induction:

$$\overline{A}(U) := \lambda x.\, A \qquad \overline{(F \times G)}(U) := \lambda x.\, \overline{F}(U)(\pi_1\, x) \times \overline{G}(U)(\pi_2\, x)$$

$$\overline{X}(U) := U \qquad \overline{(F + G)}(U) := \{\mathrm{in}_1\, x \mapsto \overline{F}(U)\, x\ ;\ \mathrm{in}_2\, y \mapsto \overline{G}(U)\, y\}$$

Moreover, given a term $f : (b : B) \to U\, b \to V\, b$ we define another term $\overline{H}(f) : (b : H[B]) \to \overline{H}(U)\, b \to \overline{H}(V)\, b$ again by induction in H:

$$\overline{A}(f) := \lambda b.\, \mathrm{id}_A \qquad \overline{(F \times G)}(f) := \lambda b.\, \overline{F}(f)(\pi_1\, b) \times \overline{G}(f)(\pi_2\, b)$$

$$\overline{X}(f) := f \qquad \overline{(F + G)}(f) := \{\mathrm{in}_1\, x \mapsto \overline{F}(f)\, x\ ;\ \mathrm{in}_2\, y \mapsto \overline{G}(f)\, y\}$$

A special case that we will use frequently is the choice $U = \mathbf{1}$, which allows us to obtain $\overline{H}(f) : (b : H[B]) \to \overline{H}(V)\, b$ from $f : (b : B) \to V\, b$. ◀

The correctness of this definition, that is, the typings announced in Definition 5 are valid, is proved by induction in the polynomial $H[X]$.

Next, we give a preparatory definition for path constructors that allow us to specify paths between two terms of the type at hand. To be able to give type-correct computation rules, these terms must be, however, of a special form, called constructor terms. Such constructor terms are built from a restricted term syntax, possibly involving the data constructors and an argument for the corresponding path constructor. We introduce constructor terms in the following definition, for which we assume the type theory introduced in Section 2.1 to be extended by the variable X as base type.

Definition 6. Let k be a positive natural number, and let $H_1, \ldots, H_k$ be polynomials and $c_1 : H_1[X] \to X, \ldots, c_k : H_k[X] \to X$ be constants. We say that r is a *constructor term (over $c_1, \ldots, c_k$)*, if there is a context Γ in which no type uses X, a variable x that does not occur in Γ, and polynomials $F[X]$ and $G[X]$, such that $x : F \Vdash r : G$ can be derived using the following rules.

$$\frac{\vdash t : A \qquad X \text{ does not occur in } A}{x : F \Vdash t : A} \qquad \frac{}{x : F \Vdash x : F} \qquad \frac{x : F \Vdash r : H_i[X]}{x : F \Vdash c_i\, r : X}$$

$$\frac{j \in \{1,2\} \qquad x : F \Vdash r : G_1 \times G_2}{x : F \Vdash \pi_j\, r : G_j} \qquad \frac{j = 1,2 \qquad x : F \Vdash r_j : G_j}{x : F \Vdash (r_1, r_2) : G_1 \times G_2}$$

$$\frac{j \in \{1,2\} \qquad x : F \Vdash r : G_j}{x : F \Vdash \mathrm{in}_j\, r : G_1 + G_2}$$

If x does not occur in r, we say that r is a *non-recursive* constructor term. ◀

Remark. We could have extended the type theory in Section 2.1 with constants $c_1, \ldots, c_k$ and use restricted terms of that theory as constructor terms. Again, to make the following development more accessible, we stick to the explicit definition given above.

We now extend MLTT with higher inductive types. To this end, we devise a scheme, whose syntax is similar to the syntax for inductive types in Coq, that allows us to introduce a new type with data constructors and path constructors. For this type we then have an elimination rule in form of dependent iteration (induction) and the corresponding computation rules. Higher inductive types that can be introduced through this scheme are of a restricted form, in that we only allow data and path constructors, but no constructors for higher paths. These are sufficient for the present exposition.

Definition 7. A *higher inductive type* is given according to the following scheme.

$$
\begin{array}{l}
\texttt{Inductive}\ T\ (B_1 : \mathrm{TYPE}) \ldots (B_\ell : \mathrm{TYPE}) := \\
\quad \mid\ c_1 : H_1[T\, B_1 \cdots B_\ell] \to T\, B_1 \cdots B_\ell \\
\quad \ldots \\
\quad \mid\ c_k : H_k[T\, B_1 \cdots B_\ell] \to T\, B_1 \cdots B_\ell \\
\quad \mid\ p_1 : (x : A_1[T\, B_1 \cdots B_\ell]) \to t_1 = r_1 \\
\quad \ldots \\
\quad \mid\ p_n : (x : A_n[T\, B_1 \cdots B_\ell]) \to t_n = r_n
\end{array}
$$

Here, all H_i and A_j are polynomials that can use $B_1, \ldots, B_\ell$, and all t_j and r_j are constructor terms over $c_1, \ldots, c_k$ with $x : A_j \Vdash t_j, r_j : T$. If X does not occur in any of the A_j, then T is called *non-recursive* and *recursive* otherwise. ◀

We now give the rules that extend the type theory given in Section 2.1 with higher inductive types, according to the scheme given in Definition 7.

Definition 8 (MLTT with HITs, Introduction Rules). For each instance T of the scheme in Definition 7, we add the following type formation rule to those of MLTT.

$$\frac{\Gamma \vdash B_1 : \textsc{Type} \qquad \cdots \qquad \Gamma \vdash B_\ell : \textsc{Type}}{\Gamma \vdash T\, B_1 \cdots B_\ell : \textsc{Type}}$$

For the sake of clarity we leave the type parameters in the following out and just write T instead of $T\,B_1 \cdots B_\ell$. The introduction rules for T are given by the following data and path constructors.

$$\frac{\vdash \Gamma \quad \textsc{Ctx}}{\Gamma \vdash c_i : H_i[T] \to T} \qquad \frac{\vdash \Gamma \quad \textsc{Ctx}}{\Gamma \vdash p_j : A_j[T] \to t_j = r_j} \qquad \blacktriangleleft$$

The dependent elimination rule for higher inductive types provides the induction principle: it allows to construct a term of type $(x : T) \to Y\,x$ for $Y : T \to \textsc{Type}$. In the hypothesis of the elimination rule we want to assume paths between elements of different types: the types $Y(t_j)$ and $Y(r_j)$. Concretely we will assume paths q as follows

$$q : (x : A) \to \widehat{t} =^{Y}_{p\,x} \widehat{r}$$

where p is the path constructor of T meaning that $p : (x : A) \to t = r$ and $\widehat{t} : Y\,t$ and $\widehat{r} : Y\,r$. We need to define $\widehat{t}$ by induction on t to state this hypothesis in the elimination rule. This is done in the following definition.

Definition 9. Let $c_i \colon H_i[X] \to X$ be constructors for T with $1 \le i \le k$ as in Definition 7. Note that each constructor term $x : F \Vdash\ r : G$ term immediately gives rise to a term $x : F[T] \vdash r : G[T]$. Given a type family $U \colon T \to \textsc{Type}$ and terms $\Gamma \vdash f_i : (x : H_i[T]) \to \overline{H}_i(U)\,x \to U(c_i\,x)$ for $1 \le i \le k$, we can define

$$\Gamma, x : F[T], h_x : \overline{F}(U)\,x \vdash \widehat{r} : \overline{G}(U)\,r$$

by induction in r as follows.

$$\begin{array}{lll} \widehat{t} := t & \widehat{x} := h_x & \widehat{c_i\,r} := f_i\,r\,\widehat{r} \\ \widehat{\pi_j\,r} := \pi_j\,\widehat{r} & \widehat{(r_1, r_2)} := (\widehat{r_1}, \widehat{r_2}) & \widehat{\mathrm{in}_j\,r} := \widehat{r} \end{array} \qquad \blacktriangleleft$$

It is straightforward to show that this definition is type correct.

Lemma 10. *The definition of $\widehat{r}$ in Definition 9 is type correct, that is, we indeed have $\Gamma, x : F[T], h_x : \overline{F}(U)\,x \vdash \widehat{r} : \overline{G}(U)\,r$ under the there given assumptions.*

We are now in the position to give the (dependent) elimination rule for higher inductive types.

Definition 11 (MLTT with HITs, Elimination and Computation). For each instance T of the scheme in Definition 7, the following dependent elimination rule is added to MLTT.

$$\frac{\begin{array}{c} Y : T \to \textsc{Type} \\ \Gamma \vdash f_i : (x : H_i[T]) \to \overline{H}_i(Y)\, x \to Y\ (c_i\ x) \qquad (\text{for } i = 1, \ldots, k) \\ \Gamma \vdash q_j : (x : A_j[T]) \to (h_x : \overline{A}_j(Y)\ x) \to \widehat{t_j} =^{Y}_{(p_j\ x)} \widehat{r_j} \qquad (\text{for } j = 1, \ldots, n) \end{array}}{\Gamma \vdash T\text{-rec}(f_1, \ldots, f_k, q_1, \ldots, q_n) : (x : T) \to Y\ x}$$

Note that $\widehat{t_j}$ and $\widehat{r_j}$ in the type of q_j depend on all the f_i through Definition 9. If all the f_i and q_j are understood from the context, we abbreviate $T\text{-rec}(f_1, \ldots, f_k, q_1, \ldots, q_n)$ to T-rec.

For every $1 \leq i \leq k$ we have a term computation rule for each $t : H_i[T]$

$$T\text{-rec}\ (c_i\ t) \longrightarrow f_i\ t\ \big(\overline{H}_i(T\text{-rec})\ t\big), \tag{2}$$

and for every $1 \leq j \leq n$ we have a path computation rule for each $a : A_j[T]$

$$\text{apd}(T\text{-rec}, p_j\ a) \longrightarrow q_j\ a\ \big(\overline{A}_j(T\text{-rec})\ a\big). \tag{3}$$

This has to be understood in the sense that we extend the reduction relation introduced in Section 2.1 with the clauses in (2) and (3), then take the compatible closure, and allow this extended reduction relation in the conversion rule (1). ◀

We can derive some simplifications of this definition for special cases of higher inductive types. First of all, if a higher inductive type T is non-recursive, then the elimination rule in Definition 11 can be simplified to

$$\frac{\begin{array}{c} Y : T \to \textsc{Type} \\ \Gamma \vdash f_i : (x : H_i[T]) \to \overline{H}_i(Y)\, x \to Y\ (c_i\ x) \qquad (\text{for } i = 1, \ldots, k) \\ \Gamma \vdash q_j : (x : A_j) \to \widehat{t_j} =^{Y}_{p_j\ x} \widehat{r_j} \qquad (\text{for } j = 1, \ldots, n) \end{array}}{\Gamma \vdash T\text{-rec}(f_1, \ldots, f_k, q_1, \ldots, q_n) : (x : T) \to Y\ x}$$

and the path computation rule becomes then

$$\text{apd}(T\text{-rec}, p_j\ a) \longrightarrow q_j\ a.$$

Second, if Y is also constant, that is, if there is $D : \textsc{Type}$ with $Y\ t \equiv D$ for all t, then we obtain the *non-dependent* elimination or *(primitive) recursion.*

$$\frac{\begin{array}{c} \Gamma \vdash f_i : H_i[T] \to H_i[D] \to D \qquad (\text{for } i = 1, \ldots, k) \\ \Gamma \vdash q_j : (x : A_j) \to \widehat{t_j} = \widehat{r_j} \qquad (\text{for } j = 1, \ldots, n) \end{array}}{\Gamma \vdash T\text{-rec}(f_1, \ldots, f_k, q_1, \ldots, q_n) : T \to D}$$

In this case, the path computation rules simplifies even further to

$$\text{ap}(T\text{-rec}, p_j\ a) \longrightarrow q_j\ a.$$

An important property of reduction relations in type theories is that computation steps preserve types of terms (*subject reduction*). To be able to show subject reduction for MLTT + HIT presented here, we need the following lemma.

Lemma 12. *Let T be a higher inductive type and T-rec an instance of Definition 11. For all constructor terms $x : F \Vdash r : G$ and terms $a : F[T]$ we have*

$$\overline{G}(T\text{-rec})\ (r[a/x]) \longrightarrow \widehat{r}\,[a/x, \overline{F}(T\text{-rec})\ a/h_x].$$

Proof. This is proved by induction in r. □

Proposition 13. *The computation rules in Definition 11 preserve types.*

Proof. That the computation rules on terms preserve types can be seen by a straightforward application of the typing rules on both sides of (2). For the computation rules on paths, on the other hand, one can derive that

$$\Gamma \vdash \mathrm{apd}(T\text{-rec}, p_j\ a) : T\text{-rec}\ (t_j[a]) =^{Y}_{p_j\ a} T\text{-rec}\ (r_j[a])$$

and

$$\Gamma \vdash q_j\ a\ \big(\overline{A_j}(T\text{-rec})\ a\big) : \widehat{t_j}[a, \overline{A_j}(T\text{-rec})\ a] =^{Y}_{p_j\ a} \widehat{r_j}[a, \overline{A_j}(T\text{-rec})\ a].$$

Using $F = A_j$ and $G = X$, we obtain from Lemma 12 that

$$\widehat{t_j}[a, \overline{A_j}(T\text{-rec})\ a] \longrightarrow T\text{-rec}\ (t_j[a]).$$

Thus, by the conversion rule, we find that $q_j\ a\ \big(\overline{A_j}(T\text{-rec})\ a\big)$ actually has the same type as $\mathrm{apd}(T\text{-rec}, p_j\ a)$. □

4 Modular Arithmetic

Modular arithmetic is not convenient to define using inductive types. One would like to imitate the inductive definition of $\mathbb{N}$ by means of constructors 0 for zero and S for the successor. However, that will always give an infinite amount of elements. If one instead defines $\mathbb{N}/m\mathbb{N}$ by taking m copies of the type $\top$ with just one element, then the definitions will be rather artificial. This way the usual definitions for addition, multiplication or other operations, cannot be given in the normal way. Instead one either needs to define them by hand, or code the $\mathbb{N}/m\mathbb{N}$ in $\mathbb{N}$ and make a map mod $m : \mathbb{N} \to \mathbb{N}/m\mathbb{N}$.

For higher inductive types this is different because one is able to postulate new identities. This way we can imitate the definition $\mathbb{N}$, and then add an equality between 0 and $S^m\,0$. However, our definition for higher inductive types does not allow dependency on terms. We can define $\mathbb{N}/2\mathbb{N}$, $\mathbb{N}/3\mathbb{N}$, and so on, but we cannot give a definition for $(m : \mathbb{N}) \to \mathbb{N}/m\mathbb{N}$. Instead of defining $\mathbb{N}/m\mathbb{N}$ in general, we thus define $\mathbb{N}/100\mathbb{N}$ which is not feasible to define using inductive types. For other natural numbers we can give the same definition.

$$\begin{array}{l}\texttt{Inductive}\ \mathbb{N}/100\mathbb{N} := \\ \quad \mid\ 0 : \mathbb{N}/100\mathbb{N} \\ \quad \mid\ S : \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N} \\ \quad \mid\ \mathrm{mod} : 0 = S^{100}\, 0\end{array}$$

This is a nonrecursive higher inductive type, because the path $0 = S^n\, 0$ does not dependent on variables of type $\mathbb{N}/100\mathbb{N}$. The definition of $\mathbb{N}/100\mathbb{N}$ gives us the constructors $0 : \mathbb{N}/100\mathbb{N}$, $S : \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N}$ and $\mathrm{mod} : 0 = S^{100}\, 0$. Furthermore, we obtain for all type families $Y : (x : \mathbb{N}/100\mathbb{N}) \to \textsc{Type}$ the following dependent recursion principle, which we refer to as induction to emphasize the relation to induction on natural numbers.

$$\frac{z : Y\, 0 \qquad s : (x : \mathbb{N}/100\mathbb{N}) \to Y\, x \to Y\, (S\, x) \qquad q : \widehat{0} =^{Y}_{\mathrm{mod}} \widehat{S^{100}\, 0}}{\mathbb{N}/100\mathbb{N}\, \mathrm{ind}(z, s, q) : (x : \mathbb{N}/100\mathbb{N}) \to Y\, x}$$

Remember $\widehat{0} =^{Y}_{\mathrm{mod}} \widehat{S^{100}\, 0}$ is defined by $\mathrm{mod}_*\ \widehat{0} = \widehat{S^{100}\, 0}$ where we define the transport mod_* using Proposition 2.

We note that, with this z and s, $\widehat{0} \equiv z$ and $\widehat{S^{100}\, 0} \equiv s\, \underline{99}\, (s\, \underline{98} \cdots (s\, 0\, z) \cdots))$, where $\underline{n}$ denotes $S^n\, 0$. Finally, we have the following computation rules

$$\mathbb{N}/100\mathbb{N}\, \mathrm{ind}(z, s, q)\ 0 \longrightarrow z,$$

$$\mathbb{N}/100\mathbb{N}\, \mathrm{ind}(z, s, q)\ (S\, x) \longrightarrow s\, x\ (\mathbb{N}/100\mathbb{N}\, \mathrm{ind}(z, s, q)\ x),$$

$$\mathrm{apd}(\mathbb{N}/100\mathbb{N}\, \mathrm{ind}(z, s, q), \mathrm{mod}) \longrightarrow q.$$

We will now demonstrate the use of the recursion principle by defining addition. To do so, we will need an inhabitant of the type $(n : \mathbb{N}/100\mathbb{N}) \to n = S^{100} n$, which means that for every $n : \mathbb{N}/100\mathbb{N}$ we have an equality of type $n = S^{100}\, n$. This can be derived from the definition of $\mathbb{N}/100\mathbb{N}$, as we demonstrate now.

Proposition 14. *There is a term* gmod: $(n : \mathbb{N}/100\mathbb{N}) \to n = S^{100}\, n$.

Proof. We define the type family $Y : \mathbb{N}/100\mathbb{N} \to \textsc{Type}$ by $\lambda n.n = S^{100}\, n$. To apply induction, we first need to give an inhabitant z of type $Y\, 0$ which is $0 = S^{100}\, 0$. Since mod is of type $0 = S^{100}\, 0$, we can take $z := \mathrm{mod}$.

Next, we have to give a function $s : (n : \mathbb{N}) \to Y\, n \to Y\, (S\, n)$, hence s must be of type $(n : \mathbb{N}) \to n = S^{100}\, n \to S\, n = S^{100}\, (S\, n)$. Thus, we can take $s := \lambda n \lambda q.\, \mathrm{ap}(S, q)$.

Finally, we need to give an inhabitant of $z =^{Y}_{\mathrm{mod}} \widehat{S^{100}\, 0}$. To do so, we first note that there is a path

$$\begin{aligned}\widehat{S^{100}\, 0} &\equiv s\, \underline{99}\, (s\, \underline{98} \cdots (s\, 0\, z) \cdots) \equiv \mathrm{ap}(S, \mathrm{ap}(S, \cdots \mathrm{ap}(s, \mathrm{mod}) \cdots)) \\ &= \mathrm{ap}(\lambda n.S^{100}\, n, \mathrm{mod}),\end{aligned}$$

where we used that for all f, g, p there is a path $\mathrm{ap}(f \circ g, p) = \mathrm{ap}(f, \mathrm{ap}(g, p))$. We can now apply Proposition 3 to $f := \mathrm{id}$, $g := \lambda n.S^{100}\, n$ and $p := q := \mathrm{mod}$ to obtain a path

$$\mathrm{mod}_*(\mathrm{mod}) = \mathrm{ap}(\mathrm{id}, \mathrm{mod})^{-1} \bullet \mathrm{mod} \bullet \mathrm{ap}(\lambda n.S^{100}\, n, \mathrm{mod}).$$

Since there is a path $\mathrm{ap}(\mathrm{id}, \mathrm{mod}) = \mathrm{mod}$, we thus obtain a path q

$$\begin{aligned}\mathrm{mod}_*(\mathrm{mod}) &= \mathrm{ap}(\mathrm{id}, \mathrm{mod})^{-1} \bullet \mathrm{mod} \bullet \mathrm{ap}(\lambda n.S^{100}\, n, \mathrm{mod}) \\ &= \mathrm{mod}^{-1} \bullet \mathrm{mod} \bullet \mathrm{ap}(\lambda n.S^{100}\, n, \mathrm{mod}) \\ &= \mathrm{ap}(\lambda n.S^{100}\, n, \mathrm{mod}) \\ &= \widehat{S^{100}\, 0},\end{aligned}$$

so that $q : z =^{Y}_{\mathrm{mod}} \widehat{S^{100}\, 0}$, and gmod is given by $\mathbb{N}/100\mathbb{N}\,\mathrm{ind}(z, s, q)$. $\square$

Using this proposition and recursion on $\mathbb{N}/100\mathbb{N}$, we can define addition as function term $+ : \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100N$. The recursion principle is, as we have shown in Section 3, a special case of induction and amounts here to

$$\frac{z : Y \qquad s : Y \to Y \qquad q : z = s^{100}\, z}{\mathbb{N}/100\mathbb{N}\text{-rec}(z, s, q) : \mathbb{N}/100\mathbb{N} \to Y}$$

with computation rules

$$\mathbb{N}/100\mathbb{N}\text{-rec}(z, s, q)\; 0 \longrightarrow z,$$

$$\mathbb{N}/100\mathbb{N}\text{-rec}(z, s, q)\; (S\, n) \longrightarrow s\; (\mathbb{N}/100\mathbb{N}\text{-rec}(z, s, q)\; n) \text{ and}$$

$$\mathrm{ap}(\mathbb{N}/100\mathbb{N}\text{-rec}(z, s, q), p) \longrightarrow q.$$

To define addition, we give for every $n : \mathbb{N}/100\mathbb{N}$ a function f_m, which represents $\lambda x.x{+}m$. So, let $m : \mathbb{N}/100\mathbb{N}$ be arbitrary, and next we define f_m using recursion. For the inhabitant z of type $\mathbb{N}/100\mathbb{N}$ we take m. Next we give a function $s : \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N}$ which will be S. Lastly, we need to give a path between m and $S^{100}\, m$, for which we can take $\mathrm{gmod}\, m$ by Proposition 14. This gives us the desired function $f_m = \mathbb{N}/100\mathbb{N}(m, S, q\, m) : \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N}$. By the computation rules we have

$$f_m\, 0 = m, \quad f_m\, (S\, x) = S\, (f_m\, x), \quad \mathrm{ap}(f_m, p) = q\, m.$$

Hence, we can define $+ : \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N} \to \mathbb{N}/100\mathbb{N}$ by the function

$$\lambda m : \mathbb{N}/100\mathbb{N} \lambda n : \mathbb{N}/100\mathbb{N}.f_m\, n.$$

5 Integers

Another interesting data type, which we will study, are the integers. These can be defined as a normal inductive type, but also as a higher inductive type. Both representations have their advantages and disadvantages. To define it as an inductive type, we can do the same as in [Licata and Shulman(2013)]. We first need to define an inductive type for the positive natural numbers. This type is called Pos and has a constructors one : Pos and $S : \text{Pos} \to \text{Pos}$.

The inductive typed definition is the same as for the natural numbers (one constant and one unary constructor), but we interpret it differently. For example, for the type Pos we define addition in a different where one + one would be S one. To clarify the distinction between the inductive types $\mathbb{N}$ we will sometimes write $S_{\mathbb{N}}$ for the successor of $\mathbb{N}$ and S_{Pos} for the successor of Pos. We have a function $i : \text{Pos} \to \mathbb{N}$ that reflects the semantics of Pos, sending one to $S_{\mathbb{N}}\,0$ and $S_{\text{Pos}}\,n$ to $S_{\mathbb{N}}\,(i\,n)$. In the reverse direction we have a function $j : \mathbb{N} \to \text{Pos}$ that reflects the semantics of Pos, sending 0 and $S_{\mathbb{N}}\,0$ to one and $S_{\mathbb{N}}\,(S_{\mathbb{N}}\,n)$ to $S_{\text{Pos}}\,(j\,(S_{\mathbb{N}}\,n))$.

Now we can define the integers. We need a constructor for zero, and we need constructors plus and minus which turn a positive number into an integer. All in all, we get the following definition.

```
Inductive ℤ1 :=
  | Z : ℤ1
  | plus : Pos → ℤ1
  | minus : Pos → ℤ1
```

We also have a recursion rule.

$$\frac{z_Y : Y \qquad \text{plus}_Y : \text{Pos} \to Y \qquad \text{minus}_Y : \text{Pos} \to Y}{\mathbb{Z}1\text{-rec}(z, \text{plus}_Y, \text{minus}_Y) : \mathbb{Z}1 \to Y}$$

If we define the integers this way, then it is possible to define functions like addition, and show that every number has an inverse. We can also show that equality is decidable.

Definition 15. A type A is said to have *decidable equality*, if the type

$$(x\,y : A) \to (x = y) + \neg(x = y)$$

is inhabited, where as usual $\neg T := T \to \mathbf{0}$ and $\mathbf{0}$ is the type with no constructors.

Proposition 16. *The type* $\mathbb{Z}1$ *has decidable equality.*

The disadvantage of this definition is that we have to redefine everything from the natural numbers to the positive numbers. Instead, one would like to define the constructors plus and minus using natural numbers. This means that we replace plus : $\text{Pos} \to \mathbb{Z}1$ by a constructor $\text{plus}' : \mathbb{N} \to \mathbb{Z}2$. However, if we define it this way, then the number 0 will be added twice. To solve this, we use higher inductive types, because then we can add equalities as well. We use almost the same definition, but in addition, we add an equality $\mathrm{plus}'\,0 = \mathrm{minus}'\,0$.

```
Inductive ℤ2 :=
  | plus′ : ℕ → ℤ2
  | minus′ : ℕ → ℤ2
  | zero : plus′ 0 = minus′ 0
```

For this type we have two constructors, namely $\mathrm{plus}' : \mathbb{N} \to \mathbb{Z}2$ and $\mathrm{minus}' : \mathbb{N} \to \mathbb{Z}2$. We also have a recursion rule.

$$\frac{\mathrm{plus}'_Y : \mathbb{N} \to Y \qquad \mathrm{minus}'_Y : \mathbb{N} \to Y \qquad \mathrm{zero}_Y : \mathrm{plus}'\ 0 = \mathrm{minus}'\ 0}{\mathbb{Z}2\text{-rec}(\mathrm{plus}'_Y, \mathrm{minus}'_Y, \mathrm{zero}_Y) : \mathbb{Z}2 \to Y}$$

The computation rules say that

$$\mathbb{Z}2\text{-rec}(\mathrm{plus}'_Y, \mathrm{minus}'_Y, \mathrm{zero}_Y)\ (\mathrm{plus}'\ n) \longrightarrow \mathrm{plus}'_Y\ n,$$

$$\mathbb{Z}2\text{-rec}(\mathrm{plus}'_Y, \mathrm{minus}'_Y, \mathrm{zero}_Y)\ (\mathrm{minus}'\ n) \longrightarrow \mathrm{minus}'_Y\ n,$$

$$\mathrm{ap}(\mathbb{Z}2\text{-rec}(\mathrm{plus}'_Y, \mathrm{minus}'_Y, \mathrm{zero}_Y), \mathrm{zero}) \longrightarrow \mathrm{zero}_Y\,.$$

Now we have two types which should represent the integers, namely $\mathbb{Z}1$ and $\mathbb{Z}2$. These types are related via an isomorphism.

Theorem 17. *We have an isomorphism* $\mathbb{Z}1 \simeq \mathbb{Z}2$.

Proof. We just show how to make the map $g : \mathbb{Z}2 \to \mathbb{Z}1$. To make the function $g : \mathbb{Z}2 \to \mathbb{Z}1$, we use the map $j : \mathbb{N} \to \mathrm{Pos}$ defined before and the recursion principle of the higher inductive type $\mathbb{Z}2$. We need to say where $\mathrm{plus}'\ n$ and $\mathrm{minus}'\ n$ are mapped to, and for that we define two functions. For the positive integers, we define $\varphi : \mathbb{N} \to \mathbb{Z}1$ which sends 0 to Z and $S_{\mathbb{N}}\,n$ to $\mathrm{plus}\,(j\ (S_{\mathbb{N}}\,n))$. For the negative integers we define the map $\psi : \mathbb{N} \to \mathbb{Z}1$ which sends 0 to Z and $S_{\mathbb{N}}\,n$ to $\mathrm{minus}\,(j\ (S_{\mathbb{N}}\,n))$. Finally, we need to give a path between $\varphi\,0$ and $\psi\,0$. Note that by definition we have $\varphi\,0 \equiv Z$ and $\psi\,0 \equiv Z$, and we choose $\mathrm{refl}\ Z$. So, we define g to be the map $\mathbb{Z}2\text{-rec}(\varphi, \psi, \mathrm{refl}\ Z)$. □

The definition of $\mathbb{Z}2$ also has a disadvantage, and to illustrate it, we try to define $+ : \mathbb{Z}2 \times \mathbb{Z}2 \to \mathbb{Z}2$. To do so, we use induction on both arguments. Now we need to give a value of $+(\mathrm{plus}'\,n, \mathrm{plus}'\,m)$ which is $\mathrm{plus}'(n+m)$. The case $+(\mathrm{minus}'\,n, \mathrm{minus}'\,m)$ is easy as well, because this is just $\mathrm{minus}'(n+m)$. However, defining $+(\mathrm{plus}'\,n, \mathrm{minus}'\,m)$ and $+(\mathrm{minus}'\,n, \mathrm{plus}'\,m)$ requires more work. We need to compare the values of n and m in order to give this. In an expression it would look like

$$+(\mathrm{plus}'\,n, \mathrm{minus}'\,m) = \text{if}\ \ n > m\ \ \text{then}\ \ \mathrm{plus}'(n-m)\ \ \text{else}\ \ \mathrm{minus}'(m-n),$$

$$+(\text{minus}'\, n, \text{plus}'\, m) = \text{if } \ n > m \ \text{ then } \ \text{minus}'(n-m) \ \text{ else } \ \text{plus}'(m-n).$$

There is also another way to represent the integers as a higher inductive type, which makes defining addition easier. The previous data types encoded the integers by partitioning them into positive and negative numbers. However, we can try to imitate the definition of the natural numbers. These have two constructors, namely 0 and the successor function S. The integers should instead have three constructors, namely 0, the successor S, and predecessor P. On top, we need to ensure that S and P are inverses, which can be achieved by using a higher inductive type as follows. As a matter of fact, this is basically the treatment of the integers that Turner gives in [Turner(1985)].

$$\begin{array}{l}
\texttt{Inductive } \mathbb{Z}3 := \\
\quad \mid \ 0 : \mathbb{Z}3 \\
\quad \mid \ S : \mathbb{Z}3 \to \mathbb{Z}3 \\
\quad \mid \ P : \mathbb{Z}3 \to \mathbb{Z}3 \\
\quad \mid \ \text{inv}_1 : (x : \mathbb{Z}3) \to P\ (S\ x) = x \\
\quad \mid \ \text{inv}_2 : (x : \mathbb{Z}3) \to S\ (P\ x) = x
\end{array}$$

For this type we have three constructors $0 : \mathbb{Z}3$, $S : \mathbb{Z}3 \to \mathbb{Z}3$, and $P : \mathbb{Z}3 \to \mathbb{Z}3$ for points, and we have two constructors $\text{inv}_1 : (x : \mathbb{Z}3) \to P\ (S\ x) = x$ and $\text{inv}_2 : (x : \mathbb{Z}3) \to S\ (P\ x) = x$ for paths. We also have a recursion rule

$$\frac{\begin{array}{ll} 0_Y : Y & \\ S_Y : Y \to Y & \text{inv}_{Y,1} : (x : Y) \to P_Y\ (S_Y\ x) = x \\ P_Y : Y \to Y & \text{inv}_{Y,2} : (x : Y) \to S_Y\ (P_Y\ x) = x \end{array}}{\mathbb{Z}3\text{-rec}(0_Y, S_Y, P_Y, \text{inv}_{Y,1}, \text{inv}_{Y,2}) : \mathbb{Z}3 \to Y}$$

This rule is derived from dependent elimination by taking the type family Y to be constant, see the discussion after Definition 11. We also get the following computation rules, where we denote $\mathbb{Z}3\text{-rec}(0_Y, S_Y, P_Y, \text{inv}_{Y,1}, \text{inv}_{Y,2})$ by $\mathbb{Z}3$-rec:

$$\mathbb{Z}3\text{-rec } 0 \longrightarrow 0_Y, \quad \mathbb{Z}3\text{-rec } (S\ x) \longrightarrow S_Y\ (\mathbb{Z}3\text{-rec } x),$$
$$\mathbb{Z}3\text{-rec } (P\ x) \longrightarrow P_Y\ (\mathbb{Z}3\text{-rec } x), \quad \text{ap}(\mathbb{Z}3\text{-rec}, \text{inv}_1\ x) \longrightarrow \text{inv}_{Y,1}\ (\mathbb{Z}3\text{-rec } x),$$
$$\text{ap}(\mathbb{Z}3\text{-rec}, \text{inv}_2\ x) \longrightarrow \text{inv}_{Y,2}\ (\mathbb{Z}3\text{-rec } x).$$

One of the interesting features of homotopy type theory is proof relevance: not all proofs of equality are considered to be equal. Let us look at the term $P\,(S\,(P\,0))$ to demonstrate this. There are two ways to prove this term equal to $P\,0$. We can use that $P\,(S\,x) = x$, but we can also use that $S\,(P\,x) = x$. Hence, we have two paths from $P\,(S\,(P\,0))$ to $P\,0$, namely $\text{inv}_1\,(P\,x)$ and $\text{ap}(P, \text{inv}_2\,(S\,(P\,0)))$. Since higher inductive types are freely generated from the points and paths, there is no reason why these two paths would be the same. As a matter of fact, one would expect them to be different which is indeed the case.

Proposition 18. *The paths* $\text{inv}_1\ (P\ (S\ (P\ 0)))$ *and* $\text{ap}(P, \text{inv}_2\ (S\ (P\ 0)))$ *are not equal.*

Before we give the proof, let us start with a proof sketch. In type theory one can prove that the empty type $\mathbf{0}$ and the type $\mathbf{1}$ with just one element, are different types. (That is, one can prove $\mathbf{0} \not\simeq \mathbf{1}$.) One can also define a type family $(n : \mathbb{N}) \to Y\ n$ sending 0 to $\mathbf{0}$ and $S\,n$ to $\mathbf{1}$. This proves that 0 and $S\,n$ are not equal. More generally, this allows us to prove that different constructors of an inductive type are indeed different.

However, for path constructors we cannot copy this argument. If we make a family of types on $\mathbb{Z}3$, then the paths inv_1 and inv_2 do not get sent to types. Hence, the induction principle cannot be used in this way to show that inv_1 and inv_2 are different. Instead we rely on the univalence axiom to prove this.

First we need a type for the circle. The definition can be given as a higher inductive type.

$\texttt{Inductive}\ S^1 :=$
$\quad |\ \mathrm{base} : S^1$
$\quad |\ \mathrm{loop} : \mathrm{base} = \mathrm{base}$

The main ingredient here is that loop and refl are unequal. One can show this by using the univalence axiom [Licata and Shulman(2013)]. To finish the proof of Proposition 18, we define a function $f : \mathbb{Z}3 \to S^1$ where the point 0 is sent to base, the maps S and P are sent to the identity. Furthermore, we send the path inv_1 to refl and inv_2 to loop. Using the elimination rule, we thus define f as $\mathbb{Z}3\text{-rec}(\mathrm{base}, \mathrm{id}, \mathrm{id}, \mathrm{refl}, \mathrm{loop})$. Note that by the computation rules, f satisfies

$$f\ 0 \longrightarrow \mathrm{base}, \quad f\ (S\ x) \longrightarrow \mathrm{id}\ (f\ x), \quad f\ (P\ x) \longrightarrow \mathrm{id}\ (f\ x),$$
$$\mathrm{ap}(f, \mathrm{inv}_1) \longrightarrow \mathrm{refl}, \quad \mathrm{ap}(f, \mathrm{inv}_2) \longrightarrow \mathrm{loop}\,.$$

Now we can finish the proof of Proposition 18.

Proof. Our goal is to show that $\mathrm{inv}_1\ (P\ (S\ (P\ 0)))$ and $\mathrm{ap}(P, \mathrm{inv}_2\ (S\ (P\ 0)))$ are not equal, for which it is sufficient to show that $\mathrm{ap}(f, \mathrm{inv}_1\ (P\ (S\ (P\ 0))))$ and $\mathrm{ap}(f, \mathrm{ap}(P, \mathrm{inv}_2\ (S\ (P\ 0))))$ are not equal. From the computation rules we get that $\mathrm{ap}(f, \mathrm{inv}_1\ (P\ (S\ (P\ 0)))) \equiv \mathrm{refl}$. One can prove by path induction that there is a path from $\mathrm{ap}(f, \mathrm{ap}(g, p))$ to $\mathrm{ap}(f \circ g, p)$ for any f and g, thus the type

$$\mathrm{ap}(f, \mathrm{ap}(P, \mathrm{inv}_2\ (S\ (P\ 0)))) = \mathrm{ap}(f \circ P, \mathrm{inv}_2\ (S\ (P\ 0)))$$

is inhabited. Using the computation rules, we see that $f \circ P$ is just f, and thus $\mathrm{ap}(f \circ P, \mathrm{inv}_2\ (S\ (P\ 0)))$ is $\mathrm{ap}(f, \mathrm{inv}_2\ (S\ (P\ 0)))$. Again we can use the computation rules, and this time it gives that $\mathrm{ap}(f, \mathrm{inv}_2\ (S\ (P\ 0))) \equiv \mathrm{loop}$. Hence, the paths $\mathrm{inv}_1\ (P\ (S\ (P\ 0)))$ and $\mathrm{ap}(P, \mathrm{inv}_2\ (S\ (P\ 0)))$ cannot be equal, because f sends them to refl and loop respectively. $\square$

Proposition 18 might not seem very interesting at first, but it actually has some surprising consequences. For that we need to use Hedberg's Theorem which says that in types with decidable equality there is only one proof of equality [Hedberg(1998)].

Theorem 19 Hedberg's Theorem. *If a type X has decidable equality, then we have a term*

$$s : (x\,y : X)\ (p\,q : x = y) \to p = q.$$

Using the contraposition from this theorem, we can thus immediately conclude that $\mathbb{Z}3$ cannot have decidable equality.

Theorem 20. *The type $\mathbb{Z}3$ does not have decidable equality.*

This sounds odd at first sight, but all it means is that we cannot decide equality just by using the induction scheme of $\mathbb{Z}3$. There are two, quite similar ways, to deal with this. We can either weaken the notion of decidable equality or we enforce that all the HITs, that are introduced through our scheme, are sets [The Univalent Foundations Program(2013)]. Let us start with the first possibility: weakening the notion of decidable equality. In (homotopy) type theory, proofs of propositions are in general relevant, in the sense that we do not just care about the existence of a proof but we are actually interested in the witness. Recall from Proposition 18 that there are two different proofs of equality between $P\ (S\ (P\ 0))$ and $P\ 0$. Thus, proof relevance prevents equality to be decidable on $\mathbb{Z}3$. However, if we reason in a proof irrelevant way by neglecting the fact that there might be several proofs for the same equality, then we obtain *merely decidable* equality. To do so, we need the so-called *truncation*, which is given by the following higher inductive type.

Inductive $||\_||\ (A : \textsc{Type}) :=$
$\mid\ \iota : A \to ||A||$
$\mid\ p : (x\,y : ||A||) \to x = y$

The truncation comes with the recursion rule

$$\frac{\iota_Y : A \to Y \qquad p_Y : (x, y : Y) \to x = y}{||A||\text{-rec}(\iota_Y, p_Y) : ||A|| \to Y}$$

and computation rules

$$||A||\text{-rec}(\iota_Y, p_Y)\ (\iota\ x) \longrightarrow \iota_Y\ x,$$

$$\text{ap}(||A||\text{-rec}(\iota_Y, p_Y), p\ x\ y) \longrightarrow p_Y\ (|A||\text{-rec}(\iota_Y, p_Y)\ x)\ (|A||\text{-rec}(\iota_Y, p_Y)\ y).$$

In the truncation every element is equal, because we add for each x, y a path $p\ x\ y$ between them. Instead of the proposition $x = y$, we can now talk about $||x = y||$. In the first type there are different proofs of equality, but in the second every element is considered to be the same. We can solve the fact that $\mathbb{Z}3$ does not have decidable equality by truncating the identity type for $\mathbb{Z}3$, as in the following theorem.

Theorem 21. *The type $\mathbb{Z}3$ has a merely decidable equality, that is: the following type is inhabited:*

$$(x\, y : \mathbb{Z}3) \to ||x = y|| + ||\neg(x = y)||.$$

Thus, if we want to consider the proofs of identities in $\mathbb{Z}3$ to be irrelevant, we have to replace the type $s = t$ by its truncation $||s = t||$ everywhere. There are two problems with that. First, this is very verbose, in that we need to introduce the truncation everywhere, and maps out of $\mathbb{Z}3$ with the truncated identity types are given in terms of the recursion principles of both $\mathbb{Z}3$ and the truncation. Second, if we want to map $\mathbb{Z}3$ with the truncated identity type to another type A, then that type must also be a set, that is, also the identity types of A may have at most one inhabitant.

Let us now study a different approach to solve the problem of decidability for $\mathbb{Z}3$. Since we do not consider higher inductive types with higher path constructors in the present setting, we are morally just dealing with quotients. However, this is not quite true, due to the fact that two paths might not be the same. For example, Theorem 20 tells us that $\mathbb{Z}3$ and $\mathbb{Z}2$ are not isomorphic. To obtain actual quotients, we need to require that each HIT definable in our setting is a set, c.f. [The Univalent Foundations Program(2013), Sec. 11.3.1]. Thus, for every HIT T defined by the scheme in Definition 7, we add a constructor

$$\mathrm{isSet}_T : (x\, y : T) \to (p\, q : x = y) \to p = q. \tag{4}$$

Note that we would also need to extend the recursion scheme for HIT in Definition 11 to account for this new constructor, since the constructor isSet_T needs to be mapped to a corresponding term in the target type. If we add for each HIT a constructor isSet_T though, then every type is a set and we can keep the original recursion scheme.

Lemma 22. *If for every higher inductive type T introduced by the scheme in Definition 7 there is a constructor* isSet_T *as in (4), then every type is a set.*

Proof. The important property of sets is that they are preserved under the type constructors in Section 2.1, see [The Univalent Foundations Program(2013), Sec 3.1 & Exerc. 3.2]. □

Forcing every HIT to be a set allows us to show that $\mathbb{Z}3$ has decidable equality. This theorem has been proved in the Coq formalization [van der Weide(2016)].

Theorem 23. *If every type is a set, then $\mathbb{Z}3 \simeq \mathbb{Z}2$ and $\mathbb{Z}3$ has decidable equality.*

6 Finite Sets

The last type we study here is a data type for finite sets. In functional programming it is difficult to work with finite sets. Often one represents them as lists on which special operations can be defined. This gives some issues in the implementation, because different lists represent the same set and the definition of a set-operation depends on the choice of the representative. For example, one could remove the duplicates or not, and depending on that choice, functions on the type will be different.

The use of higher inductive types allows to abstract from representation details. The difference between sets and lists is that in a list the order of the elements and the number of occurrences of an element matter, which does not matter for sets. Higher inductive types offer the possibility to add equalities that ignore the order of the elements and the number of occurrences. To demonstrate this, let us start by defining $\mathrm{Fin}(A)$ in a similar way as [Bauer(2016)].

```
Inductive Fin(_) (A : TYPE) :=
  | ∅ : Fin(A)
  | L : A → Fin(A)
  | ∪ : Fin(A) × Fin(A) → Fin(A)
  | assoc : (x, y, z : Fin(A)) → x ∪ (y ∪ z) = (x ∪ y) ∪ z
  | neut₁ : (x : Fin(A)) → x ∪ ∅ = x
  | neut₂ : (x : Fin(A)) → ∅ ∪ x = x
  | com : (x, y : Fin(A)) → x ∪ y = y ∪ x
  | idem : (x : A) → L x ∪ L x = L x
```

Summarizing, the type of finite sets on A is defined as the free join-semilattice on A. We abbreviate La to $\{a\}$. The constructors can be read from the definition, but we give the recursion rule and the computation rules.

$$\frac{\begin{array}{cc} & a_Y : (x, y, z : Y) \to x \cup_Y (y \cup_Y z) = (x \cup_Y y) \cup_Y z \\ & n_{Y,1} : (x : Y) \to x \cup_Y \emptyset_Y = x \\ \emptyset_Y : Y & n_{Y,2} : (x : Y) \to \emptyset_Y \cup_Y x = x \\ L_Y : A \to Y & c_Y : (x, y : Y) \to x \cup_Y y = y \cup_Y x \\ \cup_Y : Y \times Y \to Y & i_Y : (x : A) \to L_Y\, x \cup_Y L_Y\, x = L_Y\, x \end{array}}{\mathrm{Fin}(A)\text{-rec}(\emptyset_Y, L_y, \cup_Y, a_Y, n_{Y,1}, n_{Y,2}, c_Y, i_Y) : \mathrm{Fin}(A) \to Y}$$

In the following, we abbreviate $\mathrm{Fin}(A)\text{-rec}(\emptyset_Y, L_y, \cup_Y, a_Y, n_{Y,1}, n_{Y,2}, c_Y, i_Y)$ to $\mathrm{Fin}(A)$-rec. The computation rules are as follows.

$$\mathrm{Fin}(A)\text{-rec}\ \emptyset \longrightarrow \emptyset_Y, \quad \mathrm{Fin}\text{-rec}\ (L\, a) \longrightarrow L_Y\ a,$$

$$\mathrm{Fin}\text{-rec}\ (x \cup y) \longrightarrow \cup_Y\ (\mathrm{Fin}\text{-rec}\ x)\ (\mathrm{Fin}\text{-rec}\ x).$$

To demonstrate the possibilities of this definition, we define the comprehension and intersection of sets. We first define "element of a set" as a relation $\in: A \times \mathrm{Fin}(A) \to \mathrm{BOOL}$. For this relation, we need to be able to compare elements of A. This means that A must have decidable equality, so we assume that there is a term of type $(x\ y : A) \to x = y + \neg x = y$. By sending every inhabitant of $x = y$ to True and every inhabitant of $\neg x = y$ to False, we get a function $==: A \times A \to \textsc{Bool}$ which decides the equality. Using this notation we can define $\in (a, s)$ for $a : A$ and $s : \mathrm{Fin}(A)$.

Definition 24. Let A be a type with decidable equality. We define the function $\in: A \times \mathrm{Fin}(A) \to \textsc{Bool}$ by recursion on $\mathrm{Fin}(A)$ as follows.

$$\in (a, \emptyset) \equiv \mathrm{False}, \quad \in (a, \{b\}) \equiv a == b,$$

$$\in (a, x \cup y) \equiv \in (a, x) \vee \in (a, y)$$

In the notation of the recursion principle, given $a : A$ we define the function $\text{Fin-rec} : \mathrm{Fin}(A) \to \textsc{Bool}$, where we use in the recursion scheme the auxiliary functions $\emptyset_{\textsc{Bool}} := \mathrm{False}$, $\cup_{\textsc{Bool}} := \vee$, and $L_{\textsc{Bool}} := \lambda b.a == b$.

To finish the recursion, we need to give images of the paths assoc, neut_1, neut_2, com, and idem. This is not difficult to do, and we demonstrate how to do it for neut_1. We need to give an inhabitant of type $(x : \textsc{Bool}) \to x \vee \mathrm{False} = x$. That term can be given by using properties of Bool, and thus the path we choose is refl. For neut_2 we can do the same thing, and for the images of assoc, com, and idem we use that $\vee$ on Bool is associative, commutative, and idempotent. ◀

We will denote $\in (a, x)$ by $a \in x$. As seen in Definition 24, to make a map $\mathrm{Fin}(A) \to Y$, we need to give images of $\emptyset$, L, and $\cup$, and then verify some equations. Briefly said, we need to give a join semilattice Y and a map $A \to Y$. This way we also define the comprehension.

Definition 25. We define $\{ \_ \mid \_ \} : \mathrm{Fin}(A) \times (A \to \textsc{Bool}) \to \mathrm{Fin}(A)$. Let $\varphi : A \to \textsc{Bool}$. We define $\{S \mid \varphi\} : \mathrm{Fin}(A)$ by recursion on $S : \mathrm{Fin}(A)$.

$$\{\emptyset \mid \varphi\} \equiv \emptyset, \quad \{\{a\} \mid \varphi\} \equiv \text{if } \varphi\, a \text{ then } \{a\} \text{ else } \emptyset,$$

$$\{x \cup y \mid \varphi\} \equiv \{x \mid \varphi\} \cup \{y \mid \varphi\}.$$

Thus we use the recursion rule with $\emptyset_Y := \emptyset$, $L_Y\, a := \text{if } \varphi\, a \text{ then } \{a\} \text{ else } \emptyset$, and $\cup_Y := \cup$. Moreover, we to check that $\cup_Y \equiv \cup$ is associative, commutative, has $\emptyset_Y \equiv \emptyset$ as neutral element, and is idempotent. This is not difficult to check, because we have all these equalities from the constructors. ◀

Using the comprehension, we can define more operators. For example, we can define $x \cap y$ as $\{x \mid \lambda a.a \in y\}$, and $x \setminus y := \{x \mid \lambda a.\neg(a \in y)\}$.

7 Conclusion

We have given general rules for higher inductive types, both non-recursive and recursive, where we have limited ourselves to higher inductive types with path constructors. This provides a mechanism for adding data-types-with-laws to functional programming, as it provides a function definition principle, a proof (by induction) principle and computation rules. This fulfills at least partly the desire set out in [Turner(1987)] to have a constructive type theory where computation rules can be added. The use of higher inductive types and their principles was then demonstrated for typical examples that occur in functional programming. Especially the case of finite sets usually requires a considerable amount of book-keeping, which is lifted by the use of higher inductive types.

We believe that our system can be extended to include higher path constructors. This requires extending the notion of *constructor term* and extending the $\widehat{t}$ construction. It would be interesting to see which examples that arise naturally in functional programming could be dealt with using higher paths. Furthermore, it also remains to establish whether these rules are strongly normalizing, satisfy Church-Rosser and canonicity. The current definition defines HITs in type theory rather than languages like Haskell and Miranda. Hence, an open problem is incorporating HITs in Turing complete functional programming languages.

The system we have may seem limited, because we only allow constructor terms t and r in the types of equalities $t = q$ for path constructors. On the other hand, for these constructor terms we can formulate the elimination rules in simple canonical way, which we do not know how to do in general. Also, the examples we have treated (and more examples we could think of) all rely on constructor terms for path equalities, so these might be sufficient in practice.

Acknowledgments

We thank Ralf Hinze for the idea to link Turner's work and higher inductive types, and the anonymous referees for their suggestions.

References

[Abbott et al.(2005)] Abbott, M., Altenkirch, T., Ghani, N.: "Containers: Constructing strictly positive types"; Theoretical Computer Science; 342 (2005), 1, 3–27.

[Altenkirch et al.(2016)] Altenkirch, T., Capriotti, P., Dijkstra, G., Forsberg, F. N.: "Quotient inductive-inductive types"; arXiv preprint arXiv:1612.02346; (2016).

[Barthe and Geuvers(1995)] Barthe, G., Geuvers, H.: "Congruence types"; CSL; volume 1092 of Lecture Notes in Computer Science; 36–51; Springer, 1995.

[Bauer(2016)] Bauer, A.: "The Real Numbers in Homotopy Type Theory"; (2016); computability and Complexity in Analysis, 2016.

[Bauer et al.(2016)] Bauer, A., Gross, J., Lumsdaine, P. L., Shulman, M., Sozeau, M., Spitters, B.: "The HoTT Library: A Formalization of Homotopy Type Theory in Coq"; arXiv preprint arXiv:1610.04591; (2016).

[Chapman et al.(2015)] Chapman, J., Uustalu, T., Veltri, N.: "Quotienting the Delay Monad by Weak Bisimilarity"; ICTAC; volume 9399 of LNCS; 110–125; Springer, 2015.

[Gambino and Kock(2013)] Gambino, N., Kock, J.: "Polynomial functors and polynomial monads"; Math. Proc. Cambridge Phil. Soc.; 154 (2013), 01, 153–192.

[Hedberg(1998)] Hedberg, M.: "A Coherence Theorem for Martin-Löf's Type Theory"; Journal of Functional Programming; 8 (1998), 04, 413–436.

[Hofmann(1995)] Hofmann, M.: "A simple model for quotient types"; TLCA; volume 902 of Lecture Notes in Computer Science; 216–234; Springer, 1995.

[Licata and Shulman(2013)] Licata, D. R., Shulman, M.: "Calculating the Fundamental Group of the Circle in Homotopy Type Theory"; LICS; 223–232; IEEE Computer Society, 2013.

[Lumsdaine(2011)] Lumsdaine, P. L.: "Higher Inductive Types: A Tour of the Menagerie"; (2011); post on the Homotopy Type Theory Blog.

[Lumsdaine and Shulman(2012)] Lumsdaine, P. L., Shulman, M.: "Semantics of Higher Inductive Types"; Preprint; (2012).

[Martin-Löf(1975)] Martin-Löf, P.: "An Intuitionistic Theory of Types: Predicative Part"; Studies in Logic and the Foundations of Mathematics; 80 (1975), 73–118.

[Martin-Löf(1982)] Martin-Löf, P.: "Constructive Mathematics and Computer Programming"; Studies in Logic and the Foundations of Mathematics; 104 (1982), 153–175.

[Nogin(2002)] Nogin, A.: "Quotient Types: A Modular Approach"; International Conference on Theorem Proving in Higher Order Logics; 263–280; Springer, 2002.

[Nordström et al.(1990)] Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory, An Introduction; Oxford University Press (out of print now available via `www.cs.chalmers.se/Cs/Research/Logic`), 1990.

[Pierce(2002)] Pierce, B. C.: Types and Programming Languages; The MIT Press, 2002.

[Pierce(2004)] Pierce, B. C.: Advanced Topics in Types and Programming Languages; The MIT Press, 2004.

[Russell(1996)] Russell, B.: The Principles of Mathematics; WW Norton & Company, 1996.

[Sojakova(2015)] Sojakova, K.: "Higher Inductive Types as Homotopy-Initial Algebras"; POPL; 31–42; ACM, 2015.

[The Univalent Foundations Program(2013)] The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics; `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013; accessed: 2 January 2017, 10:20 am.

[Thompson(1986)] Thompson, S.: "Laws in Miranda"; Proceedings of the 1986 ACM conference on LISP and functional programming; 1–12; ACM, 1986.

[Thompson(1990)] Thompson, S.: "Lawful Functions and Program Verification in Miranda"; Science of Computer Programming; 13 (1990), 2-3, 181–218.

[Turner(1985)] Turner, D.: "Miranda: A non-strict functional language with polymorphic types"; J. Jouannaud, ed., Proceedings FPCA; volume 201 of LNCS; 1–16; 1985.

[Turner(1987)] Turner, D. N.: A new formulation of constructive type theory; University of Kent at Canterbury, Computing Laboratory, 1987.

[van der Weide(2016)] van der Weide, N.: Higher Inductive Types; Master's thesis; Radboud University; Nijmegen (2016).

[van der Weide(2016)] van der Weide, N.: "Implementation of the HIT Examples in Coq"; (2016); `https://github.com/nmvdw/HITs-Examples`.

[Whitehead and Russell(1912)] Whitehead, A. N., Russell, B.: Principia Mathematica; volume 2; University Press, 1912.