HOMOTOPY TYPE THEORY WITH AN INTERVAL TYPE

VALERY ISAEV

1. Introduction

We define a version of homotopy type theory based on a notion of the interval type. We give a simple formulation of the univalence axiom and describe a few extenisons of the system.

The first extension is called data types with conditions. Data types with conditions generalize ordinary data types, allowing us to define not only coproducts of types, but also pushouts of certain functions (which we can call cofibrations). Using data types with condtions, we will show how to define higher inductive types which have a simple description of elimination principles.

The second extension is called records with conditions and it is dual to the first one. Using this extensions, we can define path types for which the usual elimination principle J holds.

2. Syntax

In this section, we will describe the inference rules of the basic system, which has Π-types and path types. We denote by $\Leftrightarrow$ the computational equality, which is a reflexive, symmetric, and transitive closure of a union of $\Leftrightarrow_\eta$ and $\Rightarrow$, which is a union of $\Rightarrow_\beta$ and $\Rightarrow_\sigma$. Inference and reduction rules are defined in table 1.

The main ingridient of the system is the interval type I, which allows us to give a simple computational description of path types, univalence, and higher inductive types. The interval type has two constructors ($left$ and $right$) and one elimination rule (coe). The behaviour of coe can be described as follows: given a fibration $\lambda x \Rightarrow A$ over I and a point a in the fibre over $left$, $\lambda i \Rightarrow coe_{\lambda x \Rightarrow A}\ a\ i$ gives us a section of this fibration.

Path types are heterogeneous, that is we can construct paths between different types (which are itself connected by a path). We write $a =_A a'$ (or simply $a = a'$) for usual homogeneous paths, which are defined as $Path\ (\lambda i \Rightarrow A)\ a\ a'$. A path between a and a' over $\lambda x \Rightarrow A$ is a function $p : (x : I) \to A$ such that $p\ left \Leftrightarrow a$ and $p\ right \Leftrightarrow a'$. But we still can define usual functions for path types using these constructions.

Identity path (reflexivity) is a constant function $\lambda i \Rightarrow x$.

$$idp : (x : A) \to x = x$$
$$idp = \lambda x \Rightarrow path\ (\lambda i \Rightarrow x)$$

Path map (congruence) is defined in terms of a function composition. If we have a function $p : I \to A$ and a function $f : A \to B$, then $f \circ p : I \to B$ is a path

$$\frac{}{\varnothing \vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}, x \notin \Gamma \qquad \frac{\Gamma \vdash}{\Gamma \vdash x : A}, x : A \in \Gamma$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B}{\Gamma \vdash a : B}, A \Leftrightarrow B$$

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash (x : A) \to B}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x \Rightarrow b : (x : A) \to B} \qquad \frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x := a]}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash I} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathit{left} : I} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathit{right} : I}$$

$$\frac{\Gamma, x : I \vdash A \qquad \Gamma \vdash a : A[x := \mathit{left}] \qquad \Gamma \vdash j : I}{\Gamma \vdash \mathit{coe}_{\lambda x \Rightarrow A}\ a\ j : A[x := j]}$$

$$\frac{\Gamma, x : I \vdash A \qquad \Gamma \vdash a : A[x := \mathit{left}] \qquad \Gamma \vdash a' : A[x := \mathit{right}]}{\Gamma \vdash \mathit{Path}\ (\lambda x \Rightarrow A)\ a\ a'}$$

$$\frac{\Gamma, x : I \vdash a : A}{\Gamma \vdash \mathit{path}\ (\lambda x \Rightarrow a) : \mathit{Path}\ (\lambda x \Rightarrow A)\ a[x := \mathit{left}]\ a[x := \mathit{right}]}$$

$$\frac{\Gamma \vdash p : \mathit{Path}\ (\lambda x \Rightarrow A)\ a\ a' \qquad \Gamma \vdash i : I}{\Gamma \vdash p\ @_{a,a'}\ i : A[x := i]}$$

$$\begin{aligned}
&(\lambda x \Rightarrow b)\ a \Rightarrow_\beta b[x := a] \\
&\mathit{coe}_{\lambda k \Rightarrow A}\ a\ \mathit{left} \Rightarrow_\beta a \\
&\mathit{coe}_{\lambda k \Rightarrow A}\ a\ i \Rightarrow_\sigma a \text{ if } k \notin FV(A) \\
&\mathit{path}\ (\lambda x \Rightarrow t)\ @_{a,a'}\ i \Rightarrow_\beta t[x := i] \\
&p\ @_{a,a'}\ \mathit{left} \Rightarrow_\beta a \\
&p\ @_{a,a'}\ \mathit{right} \Rightarrow_\beta a' \\
&(\lambda x \Rightarrow f\ x) \Leftrightarrow_\eta f \text{ if } x \notin FV(f) \\
&\mathit{path}\ (\lambda x \Rightarrow p\ @\ x) \Leftrightarrow_\eta p \text{ if } x \notin FV(p)
\end{aligned}$$

Table 1. Inference and reduction rules.

between $f\ (p\ \mathit{left})$ and $f\ (p\ \mathit{right})$.

$$\begin{aligned}
&\mathit{pmap} : (f : A \to B) \to (a\ a' : A) \to a = a' \to f\ a = f\ a' \\
&\mathit{pmap} = \lambda f\ a\ a'\ p \Rightarrow \mathit{path}\ (\lambda i \Rightarrow f\ (p\ @\ i))
\end{aligned}$$

It is well known that functional extensionality can be defined in terms of an interval type.

$$\begin{aligned}
&\mathit{funExt} : (f\ g : (a : A) \to B[x := a]) \to ((a : A) \to f\ a = g\ a) \to f = g \\
&\mathit{funExt} = \lambda f\ g\ p \Rightarrow \mathit{path}\ (\lambda i\ a \Rightarrow p\ a\ @\ i)
\end{aligned}$$

Transporting along a path (substitution) is defined as a coercion along a path in types:

$$\begin{aligned}
&\mathit{transport} : (a\ a' : A) \to a = a' \to B[x := a] \to B[x := a'] \\
&\mathit{transport} = \lambda a\ a'\ p\ b \Rightarrow \mathit{coe}_{\lambda i \Rightarrow B[x := p\ @\ i]}\ b\ \mathit{right}
\end{aligned}$$

The definition of J is a little bit more complicated. First, we need to define a function *squeeze*, satisfying the following rules:

$$\begin{aligned}
&\mathit{squeeze} : I \to I \to I \\
&\mathit{squueze}\ \mathit{left}\ j \Rightarrow \mathit{left} \\
&\mathit{squueze}\ \mathit{right}\ j \Rightarrow j \\
&\mathit{squueze}\ i\ \mathit{left} \Rightarrow \mathit{left} \\
&\mathit{squueze}\ i\ \mathit{right} \Rightarrow i
\end{aligned}$$

The idea is that for each point on the interval $i : I$ function $\mathit{squeeze}\ i : I \to I$ maps left to left and right to i, so it squeezes the interval. Using *squeeze* we can defined a useful function *psqueeze*, which squeezes paths.

$$\begin{aligned}
&\mathit{psqueeze} : (x\ y : A)(p : x = y)(i : I) \to x = p\ @\ i \\
&\mathit{psqueeze} = \lambda x\ y\ p\ i \Rightarrow \mathit{path}\ (\lambda j \Rightarrow p\ @\ \mathit{squeeze}\ i\ j)
\end{aligned}$$

Using *psqueeze* we can define J as follows:

$$\begin{aligned}
&J : ((a : A) \to B[x := a, y := a, p := \mathit{idp}\ a]) \to (x\ y : A)(p : x = y) \to B \\
&J = \lambda d\ x\ y\ p \Rightarrow \mathit{coe}_{\lambda i \Rightarrow B[y := p\ @\ i, p := \mathit{psqueeze}\ x\ y\ p\ i]}\ (d\ x)\ \mathit{right}
\end{aligned}$$

This definition type checks since *psqueeze* satisfies the following rules:

$$\begin{aligned}
&\mathit{psqueeze}\ x\ y\ p\ \mathit{left} \Leftrightarrow \mathit{idp}\ x \\
&\mathit{psqueeze}\ x\ y\ p\ \mathit{right} \Leftrightarrow p
\end{aligned}$$

2.1. **Cubical fillers.**

The theory we are describing has many similarities to the theory of cubical sets. The reason is that we already have a 1-cube, namely the interval type, and we can define an n-cube to be the product of intervals. We do not have the product type yet, but we can form a context $x_1 : I, \ldots x_n : I$, and a term of type A in this context is simply an n-cube in A. Another reason is that *coe* gives us all n-dimensional fillers for cubical horns. We will use this to define *squeeze* in a similar way it is done in the appendix of [1].

Given an n-dimensional cubical horn we can always find a filler for it as follows:

$$\begin{aligned}
&\mathit{fill}^1_{\lambda x_1 \Rightarrow A}\ a_1\ j_1 = \mathit{coe}_{\lambda x_1 \Rightarrow A}\ a_1\ j_1 \\
&\mathit{fill}^{n+1}_{\lambda \overline{x}_{n+1} \Rightarrow A}\ (\lambda \overline{x}_n \Rightarrow a_{n+1})\ (\lambda \overline{x}_n \Rightarrow a'_{n+1}) \ldots \\
&\quad (\lambda \overline{x}_{\hat{2},n+1} \Rightarrow a_2)\ (\lambda \overline{x}_{\hat{2},n+1} \Rightarrow a'_2)\ (\lambda \overline{x}_{\hat{1},n+1} \Rightarrow a_1) = \\
&\quad \lambda \overline{j}_{n+1} \Rightarrow \mathit{fill}^n_{\lambda \overline{x}_n \Rightarrow \mathit{Path}\ (\lambda x_{n+1} \Rightarrow A)\ a_{n+1}\ a'_{n+1}} \\
&\qquad (\lambda \overline{x}_{n-1} \Rightarrow \mathit{path}\ (\lambda x_{n+1} \Rightarrow a_n))\ (\lambda \overline{x}_{n-1} \Rightarrow \mathit{path}\ (\lambda x_{n+1} \Rightarrow a'_n)) \ldots \\
&\qquad (\lambda \overline{x}_{\hat{2},n} \Rightarrow \mathit{path}\ (\lambda x_{n+1} \Rightarrow a_2))\ (\lambda \overline{x}_{\hat{2},n} \Rightarrow \mathit{path}\ (\lambda x_{n+1} \Rightarrow a'_2)) \\
&\qquad (\lambda \overline{x}_{\hat{1},n} \Rightarrow \mathit{path}\ (\lambda x_{n+1} \Rightarrow a_1))\ j_1 \ldots j_n\ @\ j_{n+1}
\end{aligned}$$

Here, $\overline{x}_k$ denotes the sequence $x_1 \ldots x_k$, and $\overline{x}_{\hat{\imath},k}$ denotes the sequence $x_1 \ldots x_{i-1}\ x_{i+1} \ldots x_k$. The idea is that a_i and a'_i are hyperfaces of an n-dimensional cube over A that form a cubical horn, and fill^n gives us a filler of this horn.

In order for this to make sense, A, a_i, and a'_i must satisfy the following rules:

$$\overline{x}_n \vdash A \qquad \overline{x}_{\hat{\imath},n} \vdash a_i : A[x_i := \mathit{left}] \qquad \overline{x}_{\hat{\imath},n} \vdash a'_i : A[x_i := \mathit{right}]$$

$$a_{i_1}[x_{i_2} := \mathit{left}] \Leftrightarrow a_{i_2}[x_{i_1} := \mathit{left}] \qquad a'_{i_1}[x_{i_2} := \mathit{left}] \Leftrightarrow a_{i_2}[x_{i_1} := \mathit{right}]$$

$$a_{i_1}[x_{i_2} := \mathit{right}] \Leftrightarrow a'_{i_2}[x_{i_1} := \mathit{left}] \qquad a'_{i_1}[x_{i_2} := \mathit{right}] \Leftrightarrow a'_{i_2}[x_{i_1} := \mathit{right}]$$

In this case fill^n satisfies the following typing rule:

$$\begin{aligned}
&\vdash \mathit{fill}^n_{\lambda\overline{x} \Rightarrow A}\ (\lambda\overline{x}_{n-1} \Rightarrow a_n)\ (\lambda\overline{x}_{n-1} \Rightarrow a'_n) \ldots \\
&\quad (\lambda\overline{x}_{\hat{2},n} \Rightarrow a_2)\ (\lambda\overline{x}_{\hat{2},n} \Rightarrow a'_2)\ (\lambda\overline{x}_{\hat{1},n} \Rightarrow a_1) \\
&: (\overline{j}_n : I) \to A[x_1 := j_1] \ldots [x_n := j_n].
\end{aligned}$$

If j_k is equal to left or right for some k, then fill^n on $j_1 \ldots j_n$ equals to $a_k[x_1 := j_1] \ldots [x_n := j_n]$ or $a'_k[x_1 := j_1] \ldots [x_n := j_n]$ respectively. So fill^n indeed returns an n-cube with given hyperfaces.

Now, we can define function *squeeze* which satisfies the required rules. There rules can be stated as follows. We need to find a function $\mathit{squeeze} : I \to I \to I$, which is given on borders by the following diagram:

$$\begin{tikzcd}
\mathit{left} \arrow[r, "\lambda i \Rightarrow \mathit{left}"] \arrow[d, "\lambda j \Rightarrow \mathit{left}"'] & \mathit{left} \arrow[d, "\lambda j \Rightarrow j"] \\
\mathit{left} \arrow[r, "\lambda i \Rightarrow i"'] & \mathit{right}
\end{tikzcd}$$

We can define function *sq*, which satisfies three of the four rules, by filling the following horn:

$$\begin{tikzcd}
\mathit{left} \arrow[r, "\lambda i \Rightarrow \mathit{left}"] \arrow[d, "\lambda j \Rightarrow \mathit{left}"'] & \mathit{left} \arrow[d, "\lambda j \Rightarrow j"] \\
\mathit{left} & \mathit{right}
\end{tikzcd}$$

$$\begin{aligned}
&sq : I \to I \to I \\
&sq = \mathit{fill}^2_{\lambda i\ j \Rightarrow I}\ (\lambda j \Rightarrow \mathit{left})\ (\lambda j \Rightarrow j)\ (\lambda i \Rightarrow \mathit{left})
\end{aligned}$$

Now, we can construct *squeeze*, which satisfies all of the rules, by filling the following horn:

$$\begin{tikzcd}
\mathit{left} \arrow[rrr] \arrow[ddd] & & & \mathit{left} \arrow[ddd] \\
& \mathit{left} \arrow[ul] \arrow[r] \arrow[d] & \mathit{left} \arrow[ur] \arrow[d] & \\
& \mathit{left} \arrow[dl] \arrow[r] & \mathit{left} \arrow[dr] & \\
\mathit{left} \arrow[rrr] & & & \mathit{right}
\end{tikzcd}$$

The inner, left, and top squares are $\lambda i\ j \Rightarrow left$, the bottom and right squares are sq, and the filler gives us the outter square which is the required function $squeeze$.

$squeeze : I \to I \to I$

$squeeze = fill^3_{\lambda x_1\ x_2\ x_3 \Rightarrow I}\ (\lambda x_1\ x_2 \Rightarrow left)\ sq\ (\lambda x_1\ x_3 \Rightarrow left)\ sq\ (\lambda x_2\ x_3 \Rightarrow left)\ right$

2.2. **Univalent universes.** Now, we show how to add a univalent universe to the system.

$$\frac{\Gamma \vdash}{\Gamma \vdash Type} \qquad \frac{\Gamma \vdash}{\Gamma \vdash I : Type} \qquad \frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash B : Type}{\Gamma \vdash (x : A) \to B : Type}$$

$$\frac{\begin{array}{llll} \Gamma \vdash A : Type & \Gamma \vdash f : A \to B & \Gamma \vdash p : (a : A) \to g\ (f\ a) = a & \\ \Gamma \vdash B : Type & \Gamma \vdash g : B \to A & \Gamma \vdash q : (b : B) \to f\ (g\ b) = b & \Gamma \vdash i : I \end{array}}{\Gamma \vdash iso\ A\ B\ f\ g\ p\ q\ i : Type}$$

We also add the following reduction rules:

- $iso\ A\ B\ f\ g\ p\ q\ left \Rightarrow_\beta A$
- $iso\ A\ B\ f\ g\ p\ q\ right \Rightarrow_\beta B$
- $iso\ A\ B\ (\lambda x \Rightarrow x)\ (\lambda x \Rightarrow x)\ idp\ idp\ i \Rightarrow_\beta A$
- $coe_{\lambda k \Rightarrow iso\ A\ B\ f\ g\ p\ q\ k}\ a\ right \Rightarrow_\beta f\ a$ if $k \notin FV(A\ B\ f\ g\ p\ q)$

Usually, univalence is stated in the form of an axiom as follows:

$$isContr\ A = \Sigma\ (a : A)\ ((a' : A) \to a = a')$$

$$isEquiv\ f = (b : B) \to isContr\ (\Sigma\ (a : A)\ (f\ a = b))$$

$$pte : (A\ B : Type) \to A = B \to \Sigma\ (f : A \to B)\ (isEquiv\ f)$$
$$pte = \lambda A\ B\ p \Rightarrow (transport\ A\ B\ p, \_)$$

$$univalence = (A\ B : Type) \to isEquiv\ (pte\ A\ B)$$

Instead of $\_$ there should be a proof that $transport\ A\ B\ p$ is an equivalence, which we omit because of its length.

We can show that $univalence$ holds using iso. First, we construct the following function

$$etp : (A\ B : Type) \to \Sigma\ (f : A \to B)\ (isEquiv\ f) \to A = B$$
$$etp = \lambda A\ B\ p \Rightarrow path\ (\lambda i \Rightarrow iso\ A\ B\ p.proj_1\ \_\ \_\ \_\ i)$$

Omitted terms can be constructed using a proof of $isEquiv\ f$.

To prove that $pte\ A\ B$ is an equivalence it is enough to show that $etp\ A\ B$ is its inverse. The first components of pairs $pte\ A\ B\ (etp\ A\ B\ (f, e))$ and (f, e) are equal definitionally, and its second components are equal since $isEquiv\ f$ is a proposition. Finally, we need to show that for each $p : A = B$ we have a path $etp\ A\ B\ (pte\ A\ B\ p) = p$. After applying J we only need to show this for $p = idp\ A$, but this holds definitionally.

2.3. **Data types with conditions.** In this section we describe, we will describe an extension of usual inductive data types, which allows us to define higher inductive types. To do this, we need to describe a way of defining functions over inductive data types. Traditionally such functions are defined either by pattern matching or through eliminators. The former is more convenient when working inside the theory, but the latter is easier to describe formally. Our approach combines both of these methods.

To allow definitions of data types and functions we extend the system by introducing the notion of signature. A signature consists of declarations of data types and functions. We need to modify our typing judgements:

- $\Sigma; \Gamma \vdash$ means that Γ is well typed context in signature Σ.
- $\Sigma; \Gamma \vdash A$ means that A is a well typed type in context Γ and signature Σ.
- $\Sigma; \Gamma \vdash a : A$ means that a is a well typed term of type A in context Γ and signature Σ.

We omit Σ from the notation if it is clear which signature we are using.

An inductive data type D with parameters $a_1 : A_1, \ldots a_n : A_n$ is described by a list of its constructors, and for each constructor a list of types of its arguments:

$$c_1\ (x_1 : B^1_1)\ \ldots\ (x_{k_1} : B^1_{k_1})$$

$$\vdots$$

$$c_m\ (x_1 : B^m_1)\ \ldots\ (x_{k_m} : B^m_{k_m})$$

We require data types to be *strictly positive*, which means that if D appears in B^i_j, then $B^i_j = F(D\ a_1\ \ldots\ a_n)$ for some $a_1, \ldots a_n$ and some strictly positive F. A function F is strictly positive if it is inductively generated by the following rules:

- $F(Z) = Z$
- $F(Z) = Path\ (\lambda i \Rightarrow F(Z))\ z_1\ z_2$
- $F(Z) = (e : E) \to F(Z)$

We say that a data type is *simple* if it doesn't use the last rule in the description of F. Of course, a data type definition must be well typed, which means that the following holds:

- $\Sigma; a_1 : A_1. \ldots a_n : A_n \vdash$.
- $\Sigma, D'; x_1 : B^i_1, \ldots x_{j-1} : B^i_{j-1} \vdash B^i_j$ for each constructor c_i and each $1 \leq j \leq k_i$, where D' is a data type with the same parameters as D, but without constructors.

A function definition consists of a name of the function with its type and a body of the function:

$$f : (x_1 : A_1) \ldots (x_n : A_n) \to B$$
$$f\ x_1\ \ldots\ x_n \Rightarrow b$$

The definition is well typed if $x_1 : A_1, \ldots x_n : A_n \vdash b : B$. We also introduce another way of defining a function, which is writtern like this:

$$f' : (x_1 : A_1) \ldots (x_n : A_n) \to B$$
$$f'\ x_1\ \ldots\ x_n \Leftarrow b$$

Such definitions are well typed if the following holds:

$$x_1 : A_1, \ldots x_n : A_n \vdash b : B$$

The difference between these definitions lies in the computation rules. For the former definition we add usual computation rules:

$$f\ a_1\ \ldots\ a_n \Rightarrow_\beta b[x_1 := a_1]\ldots[x_n := a_n]$$

For the latter computation rules are defined by the following inductive rule:

$$\frac{b[x_1 := a_1]\ldots[x_n := a_n] \Rightarrow_t b'}{f'\ a_1\ \ldots\ a_n \Rightarrow_\beta b'}$$

where $\Rightarrow_t$ is the union of the basic reduction rules that we gave in table 1 and here for function definitions. That is, $\Rightarrow_t$ makes reductions only on the top level of the term, but not inside it. Such definitions are useful when the right hand side is a big term, which usually performs some sort of case analysis by pattern matching, and we want to reduce a function application only when this case analysis is satisfied.

To allow definitions by pattern matching we introduce new primitive operator *elim*:

$$\mathit{elim}\ e\ \{\ c_1\ x_1\ \ldots\ x_{k_1} = b_1;\ \ldots;\ c_m\ x_1\ \ldots\ x_{k_m} = b_m\ \},$$

where each of the $=$ signs is either $\Rightarrow$ or $\Leftarrow$, x_i are variables, c_i are constructors of some data type, and e and b_i are some terms. The set of rules inside $\{\ \}$ is unordered, and each constructor c_i occurs exactly once. Reduction rules are the following:

$$\frac{b_i[x_1 := a_1]\ldots[x_n := a_n] \Rightarrow_t b'}{\mathit{elim}\ c_i\ a_1\ \ldots\ a_{k_i}\ \{\ c_i\ x_1\ \ldots\ x_{k_i} \Leftarrow b_i;\ \ldots\ \} \Rightarrow_\beta b'},$$

$$\mathit{elim}\ c_i\ a_1\ \ldots\ a_{k_i}\ \{\ c_i\ x_1\ \ldots\ x_{k_i} \Rightarrow b_i;\ \ldots\ \} \Rightarrow_\beta b_i[x_1 := a_1]\ldots[x_{k_i} := a_{k_i}].$$

Inference rules for *elim* are the following:

$$\frac{\begin{array}{l}\Gamma, y : D\ p_1\ \ldots\ p_n \vdash B \\ \Gamma \vdash e : D\ p_1\ \ldots\ p_n \\ \Gamma, x_1 : A^i_1, \ldots x_{k_i} : A^i_{k_i} \vdash b_i : B[y := c_i\ x_1\ \ldots\ x_{k_i}] \text{ for each } 1 \le i \le m\end{array}}{\Gamma \vdash \mathit{elim}\ e\ \{\ c_1\ x_1\ \ldots\ x_{k_1} = b_1;\ \ldots;\ c_m\ x_1\ \ldots\ x_{k_m} = b_m\ \} : B[y := e]},$$

where $c_i\ (x_1 : A^i_1)\ \ldots\ (x_{k_i} : A^i_{k_i})$ are constructors of data type D.

To allow recursive definitions we introduce new context, which consists of recursive calls to the function we are defining. Now, judgements look like this $\Gamma; \rho \vdash a : A$. Here, Γ is the usual context, and ρ is the recursive calls context. A recursive function f as before is well typed if the following holds:

$$x_1 : A_1, \ldots x_n : A_n; f\ x_1\ \ldots\ x_n : B \vdash b : B$$

We add the following inference rule, which allows us to use recursive calls:

$$\frac{}{\Gamma; t : T, t_1 : T_1, \ldots t_n : T_n \vdash t_i : T_i}$$

The first entry $t : T$ is always a recursive call to the function with the same arguments, so we do not allow to use it.

We add inference rules for *elim*, which extends recursive calls context appropriately:

$$\frac{\Gamma \vdash B \qquad \Gamma'_i; \rho'_i \vdash b_i : B[z_j := c_i\ x_1\ \ldots\ x_{k_i}] \text{ for each } 1 \le i \le m}{\Gamma; \rho \vdash \mathit{elim}\ z_j\ \{\ c_1\ x_1\ \ldots\ x_{k_1} = b_1;\ \ldots;\ c_m\ x_1\ \ldots\ x_{k_m} = b_m\ \} : B},$$

where z_j is a variable in context Γ, and contexts Γ'_i and ρ'_i are defined as follows:

- If $\Gamma = z_1 : Z_1, \ldots z_s : Z_s$, then $\Gamma'_i = z_1 : Z_1, \ldots z_{j-1} : Z_{j-1}, x_1 : A^i_1, \ldots x_{k_i} : A^i_{k_i}, z_{j+1} : Z_{j+1}[z_j := c_i\ x_1\ \ldots\ x_{k_i}], \ldots z_s : Z_s[z_j := c_i\ x_1\ \ldots\ x_{k_i}]$.
-

A data type with conditions also allows us to put a condition on a constructor. A condition on a constructor is simply a partial definition of a function with the type of this constructor. Syntactically, this means that constructors now can be evaluated if its arguments match one of the clauses of this partially defined function. Also, this means that when we define a function over such a data type we need to check that this definition respects conditions.

Semantically, ordinary data types allow to define coproducts of types, and data types with conditions allow us to define pushouts. One of the maps in the pushout diagram must be given by a collection of constructors of a data type, which means it is (often can be interpreted as) a cofibration. This implies that this pushout is allways a correct homotopy pushout.

Let us give an example. We can define integers as the data type with two constructors:

$$\mathit{negative} : \mathbb{N} \to \mathbb{Z}$$

$$\mathit{positive} : \mathbb{N} \to \mathbb{Z}.$$

The problem is that we get two different zeros: *negative* 0 and *positive* 0. Of course, we can define -1 to be *negative* 0, -2 to be *negative* 1, and so one, but this can easily lead to a confusion. It is better to identify the positive and the negative zeros, that is to define $\mathbb{Z}$ as the pushout of $1 \xrightarrow{0} \mathbb{N}$ with itself. So, we add the following condition to the definition of $\mathbb{Z}$:

$$\mathit{negative}\ 0 = \mathit{positive}\ 0.$$

Now, when we define a function f over $\mathbb{Z}$, we need to check that $f\ (\mathit{negative}\ 0) \Leftrightarrow f\ (\mathit{positive}\ 0)$.

Using data types with conditions, we can define higher inductive types. For example, the circle S^1 can be defined as follows: it has two constructors

$$\mathit{base} : S^1$$

$$\mathit{loop} : I \to S^1$$

and two conditions

$$\mathit{loop}\ \mathit{left} = \mathit{base}$$

$$\mathit{loop}\ \mathit{right} = \mathit{base}.$$

After we add path types, it will be posible to describe elimination rules for such data types, but it is more convenient to simply define functions over them.

2.4. **Records with conditions.** Records with conditions are dual to data types with conditions. Ordinary records allow us to define product of types, and records with conditions allow us to define pullbacks. A record is given by a list of fields, and a record with conditions can additionally put conditions on some fields. A condition on a field is a partially defined function with the type of this field. Semantically, such a record is given by a pullback of a function $(B \to X) \to (A \to X)$ for some cofibration $A \to B$, which means that it is still fibrant.

Let us show how to define path types as a record with conditions. It will be a record *Path* with three parameters $A : I \to \mathit{Type}$, $a : A\ \mathit{left}$, and $a' : A\ \mathit{right}$. We will define a "heterogeneous path type" because it seems impossible to define useful elimination rules for higher inductive types using only homogeneous path types. We abbreviate $Path\ (\lambda i \Rightarrow A)\ a\ a'$ to $a = a'$. The record $Path\ (A : I \to Type)\ (a : A\ left)\ (a' : A\ right)$ has one constructor

$$path : ((i : I) \to A\ i) \to Path\ (A : I \to Type)\ (a : A\ left)\ (a' : A\ right),$$

one field

$$at : (i : I) \to A\ i,$$

and two conditions

$$at\ left = a$$
$$at\ right = a'.$$

When we define an element of a record with conditions, we need to check that it satisfies them. And when we access fields with conitions, they might evaluate, if arguments to it match some condition on it. For example, if $p : a = a'$, then $p.at\ left$ evaluates to a, $p.at\ right$ evaluates to a', and if we want to consruct a path of type $a = a'$, then we need to specify a function $f : I \to A$ such that $f\ left \Leftrightarrow a$ and $f\ right \Leftrightarrow a'$.

We can define the J rule as follows:

$$\begin{aligned}
&idp : (A : Type)(a : A) \to Path\ (\lambda i \Rightarrow A)\ a\ a\\
&idp\ A\ a = path(\lambda i \Rightarrow a)\\
&\\
&J : (A : Type)(C : (x\ y : A) \to x = y \to Type) \to\\
&\qquad ((a : A) \to C\ a\ a\ (idp\ A\ a)) \to (a\ a' : A)(p : a = a') \to C\ a\ a'\ p\\
&J\ A\ C\ d\ a\ a'\ p = \ coe_{\lambda i \Rightarrow C\ a\ (p.at\ i)\ (path\ (\lambda j \Rightarrow p.at\ (squeeze\ i\ j)))}\ left\ (d\ a)\ right
\end{aligned}$$

Now, we can prove the univalence axiom in its usual form, which states that a certain function F is equivalence. Function F maps path types $A = B$ to the type of functions $A \to B$ which are equivalences. We can construct an inverse G to function F: given an equivalence $A \to B$, G construct a path $A = B$ using *iso*. If $f : A \to B$ is an equivalence, then $F\ (G\ f)$ definitionally equals to f because of the compuation rule for *iso*. If $p : A = B$, then we can show that there is a path between $G\ (F\ p)$ and p using J. Actually, we can't use J directly because of the size issues, but we can expand its definition, and then it works fine.

3. A model of data types and records with conditions

In this section, we will show how to interpret data types and records with conditions in simplicial sets.

3.1. Data types with conditions.

Each constructor of a data type gives us an accessible functor $C_i : \mathbf{sSet} \to \mathbf{sSet}$.

3.2. Records with conditions.

References

1. Marc Bezem, Thierry Coquand, and Simon Huber, *A model of type theory in cubical sets*, (2014).