First year report

Ambrus Kaposi

November 2013

Contents

1 Introduction

Computer scientists use a multitude of ways for ensuring program correctness: testing programs with different inputs and rejecting them if they do not return the expected output; monitoring the behaviour of the program during runtime; modelling the program behaviour and (exhaustively) checking that the model takes the correct steps in all states of the world; rejecting invalid programs according to a specification in a type system; (formally) proving programs correct according to a semantics of the programming language.

Type systems are popular among these methods because they easily fit into the program development process by building a type checker into the compiler. The type of a program can be viewed as a lightweight specification, the validation of which is ensured automatically by the compiler.

If the type system is not expressive enough it prohibits abstraction: in a simply typed language, one needs to define a separate `length` function for lists of integers, for lists of booleans, etc. The type systems of state of the art functional programming languages Haskell [55] and ML [51] are built on the polymorphic lambda calculus System F [28] which allows the `length` function to be written once and for all for all list types. Haskell's type system evolved naturally during the years and now it is able to express the types of ordered lists or numbers less than a given number [40], coming close to being able to express any possible specification: a dependent type system.

Dependent type theories have been proposed as the foundations of mathematics by viewing types as theorems and terms of a given type as proofs of the particular theorem. They have the expressivity needed for doing mathematics, and since all of our specifications are expressed in mathematics, they are the ultimate candidates for a type system. However, the exact choice of which dependent type theory to use is not clear. One candidate is Martin-Löf's type theory (MLTT) [43]. The programming languages Agda [53] and Coq [46] are two implementations of variants of this type theory. However, they lack the justification of some informal practices in mathematics such as considering point-wise equal functions equal or isomorphic objects interchangeable. Observational Type Theory [5] takes one step forward by identifying point-wise equal functions. Homotopy type theory [58] is a more recent candidate which validates all of these practices but lacks a full computational interpretation which prevents us from using it as a programming language. My aim is to help developing this computational interpretation.

1.1 Structure

In what follows, in section 2.1 we present Martin-Löf type theory informally as mathematical foundations in which all later definitions should be understood. I am interested in formalizing type theory inside type theory, this is why I introduce the syntax in a low-level way (e.g. explicit substitutions, Universes à la Tarski) which is more convenient to formalise. At the same time, we explain a higher level notation which can be used later when we use type theory as a metatheory. In section 2.2 we introduce some basic notions of category theory to be used as a language, then we present the additional rules of homotopy type theory (section 2.3). A way to study type theories is looking at models — in section 3 we define a categorical notion of model and then start spelling out

Thierry Coquand's Kan semisimplicial set model which validates the additional rules of homotopy type theory. I conclude with further plans and a section about other topics that I studied during my first year (section 5).

I make no claims of originality in this text.

2 Technical background

2.1 Martin-Löf Type Theory

Type theory is a formal system enabling the derivation of certain kinds of judgments. To define a type theory, one needs to list the kinds of judgments together with their syntax and list the derivation rules one could use to build derivation trees (proof trees). The root of a derivation tree is the judgment which was derived by the tree, while the leaves are axioms (derivation rules having no hypotheses). One derivation tree represents a proof of the theorem represented by the judgment at the root. All proofs in informal (constructive) mathematics should be representable by a derivation tree in type theory. Our presentation also tries to give a minimal set of rules which could serve as the core of a programming language / proof assistant based on type theory.

2.1.1 Intuition

Before listing the derivation rules in Martin Löf Type Theory, I give some intuition explaining the syntax in 4 steps.

1. (Contexts.) To represent implication, we need a way to express its introduction rule:

$$\frac{\begin{matrix}[A]\\ B\end{matrix}}{A \to B} \to \text{-intro, discharging assumption [A]}$$

   The $[A]$ in the top means that the usage of A is allowed in the proof of B, however A is only in scope in this proof of B. By discharging it when deriving $A \to B$, we cannot make use of A anymore (A is bound at the point of discharging it and is only in scope above it). We could extend the definition of proof tree to that of binding tree (see [30], chapter 1.2), but instead we introduce *contexts*: each judgment begins with a context which lists the non-discharged assumptions that are currently usable. The above introduction rule is formulated by removing the appropriate assumption from the context:

$$\frac{\Gamma.A \vdash B}{\Gamma \vdash A \to B} \to \text{-intro}$$

   If there is no assumption we write $() \vdash A$ where $()$ represents the empty context.

2. (Terms and variable binding.)

   Type theory provides a way for representing proof trees inside judgments. It defines a language of terms, each term representing one proof tree. The judgment $\Gamma \vdash t : T$ means that assuming Γ, the term t represents a proof tree with T at its root. When listing the rules of type theory we always define term formers representing the proof step of applying the particular rule. On one hand this is a concise notation for proofs (as used in proof assistants), on the other hand terms can be viewed as programs. E.g. a term of type $A \to B$ is a program which takes an input of type A and outputs something of type B. Reconstructing the proof tree from a term is called type inference. In a programming language based on type theory such as Agda, terms usually do not include all the information needed to reconstruct the proof tree, but they need to be given together with their types, and from the type and the term the proof tree can be reconstructed (this is called type checking). In our informal presentation we also omit much information from the notation for terms in order to make the notation more readable.

For representing the $\to$ -intro rule as a term, we introduce the last assumption of the context inside the term by the *binder* λ. It binds a variable representing the assumption within the term. The term represents the part of the proof tree above the use of the $\to$ -intro rule where the assumption can be used, this is the scope of the variable. There are two ways of expressing variable binding within a linear syntax:

- Introducing a new, previously unused variable name at each λ abstraction. In $\lambda x.\lambda y.x$, the x refers to the outer binding. If viewed as a program, this term represents the constant function.
- Instead of introducing variables by names, use natural numbers to refer to the binders (de Bruijn indexing). 0 refers to the innermost binder, 1 refers to the next binder etc. The previous example becomes becomes $\lambda.\lambda.1$.

In the latter case one does not need to care about α-renaming (renaming of variables) to define an equality on terms.

It is useful to generalize the above two ways to reference assumptions within the context (terms having such references are called open terms):

- give each assumption a unique name different from the bound names in the term, e.g. $z : A, z' : B, z'' : C \vdash t : D$ where t can contain variable names z, z' or z''. If one of these variables is bound within t then it shadows the binding by the context.
- refer to the assumptions by natural numbers n, $n+1$, $n+2$, etc, n being the number of binders under which the reference is placed inside the term. n would refer to the rightmost assumption, e.g. $A.B.C \vdash \lambda.2 : B$ means $z : A, z' : B, z'' : C \vdash \lambda x.z' : B$.

3. (Machinery for substitution.) We would like to equate certain proof trees representing essentially the same proofs. In particular, cut elimination should be true, i.e. introducing an implication $A \to B$ and then immediately eliminating it by modus ponens by providing an A should be the same as substituting all the occurrences of A in the proof of $A \to B$ by

the proof of A provided. When representing this by terms instead of trees we get the following rule[1]:

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{app}(\lambda x.t, a) \equiv t[a/x] : B}$$

app represents the usage of modus ponens, $t[a/x]$ is a meta-notation for the term t where every occurrence of x is replaced by a. Another way of representing substitution is building it into our theory as a primitive notion i.e. having a kind of judgment $\sigma : \Delta \to \Gamma$ which expresses that σ is a substitution from one context to the other. If $\Gamma = A_1 \dots A_n$, then σ is a list of terms $t_1 : A_1, \dots, t_n : A_n$ and these terms have free variables in Δ. This kind of syntactic presentation is called *explicit substitution.* Applying a substitution $\sigma : \Delta \to \Gamma$ on a term $\Gamma \vdash t : A$ results in a term $\Delta \vdash t\sigma : A$.

Given a special substitution $\mathsf{p} : \Gamma.A \to \Gamma$ and a special term $\Gamma.A \vdash \mathsf{q} : A$, we can represent all variables by the de Bruijn method: 0 is represented by q, 1 is represented by qp, 2 is represented by qpp etc. If $\Gamma \vdash a : A$ and $\Gamma.A \vdash t : B$ then instead of $t[a/x]$[2] we write $t(1, a)$ where $(1, a) : \Gamma \to \Gamma.A$ is a substitution built up from the identity substitution $1 : \Gamma \to \Gamma$ and the term a.

4. (Dependent types.)

   In dependent type theories types can depend on terms, terms and types are in the same syntactic category[3]. This adds some twists to the above explanations. A type now depends on a context and we use the notation $\Gamma \vdash A$ meaning that the type A might include free variables in Γ. However it is the same notation as previously where $\Gamma \vdash A$ meant A is derivable from the assumptions in Γ (and later we have introduced a proof term t for describing such a derivation: $\Gamma \vdash t : A$), the meaning is different: we have no proof terms in the typing judgements for types. With this in mind, $A_1 \dots A_n$ being a context entails the following judgments:

$$\begin{aligned} () &\vdash A_1 \\ A_1 &\vdash A_2 \\ A_1.A_2 &\vdash A_3 \\ &\dots \\ A_1 \dots A_{n-1} &\vdash A_n \end{aligned}$$

   That is, a type in a context can include variables in the preceding part of the context. A substitution $\sigma : \Gamma \to A_1 \dots A_n$ can be thought of as a list

[1]The relationship between cut elimination and β-reduction is subtle and is described in detail in [9].

[2]x is not mentioned here as a free variable in t, so $t[a/x]$ does not even make sense.

[3]In our presentation, they are in different syntactic category, however, with a universe, one can give codes for types as terms, see section 2.1.8

of terms

$$\begin{aligned}
&\Gamma \vdash t_1 : A_1\\
&\Gamma \vdash t_2 : A_2(1, t_1)\\
&\Gamma \vdash t_3 : A_3((1, t_1), t_2)\\
&\dots\\
&\Gamma \vdash t_n : A_n(\dots((1, t_1), t_2), \dots), t_{n-1})
\end{aligned}$$

A type B which depends on a context A written as $x : A \vdash B$ can be viewed as a family of types indexed over A. That is, if $a_0 : A$, then $B[a_0/x]$ is a member of the family (at index a_0). If $a_0 : A$, then $B[a_1/x]$ is another member (at index a_1).

We present two notations for the syntax of type theory, one high level notation for usage as a metatheory and one low level to be formalised.

The meta level presentation uses names for variables, the usual $f(g)$ notation for function application, pattern matching or just informal language for defining functions. We will use this as the notation for the metalanguage to distinguish it from the notation for the object theory which we will try to formalise. We only need this because the object theory will be the same as the metatheory (with some extensions).

Our main emphasis is on the lower level presentation which uses explicit substitutions for variable bindings. De Bruijn indices are simulated with the special term q (corresponding to 0) and multiple applications of the substitution p (corresponding to the successor function). Most of the time we follow the notation used by Thierry Coquand in [10]. Function application uses the app construct, universes are given by codes reflected in types by El, eliminators are used instead of pattern matching (for a while), and term formers are not automatically curried functions but have fixed arities. Sometimes we repeat definitions with variable names just for readability.

2.1.2 Kinds of judgments

We have 8 kinds of judgments:

$\Gamma \vdash$	Γ is a valid context
$\sigma : \Delta \to \Gamma$	σ is a substitution from context Δ to Γ
$\Gamma \vdash A$	A is a type in context Γ
$\Gamma \vdash t : A$	t is a term of type A in context Γ

The next four kinds of judgments express equality for the above constructs: contexts, substitutions, types and terms.

$\vdash \Gamma \equiv \Delta$	contexts Γ and Δ are definitionally equal
$\sigma \equiv \delta : \Delta \to \Gamma$	substitutions σ and δ are definitionally equal
$\Gamma \vdash A \equiv B$	types A and B in context Γ are definitionally equal
$\Gamma \vdash u \equiv v : A$	terms u and v of type A in context Γ are definitionally equal

Usually we will omit the context and type from the equality judgments and just write equations like $t \equiv r$.

The equality $\equiv$ is called convertibility or judgmental equality. In intensional Martin-Löf Type Theory, this judgmental equality expresses definitional equality: an equality relation which is defined by abbreviatory definitions (we use the $:\equiv$ symbol for these) and by the rule that substituting equals for equals should be equal.

Reasoning inside type theory means *defining* terms by the term introduction rules given below. From within type theory, we only have access to term formers (and substitutions, which we use instead of variables), and we cannot access contexts, types and definitional equality. The terms that we define by using $:\equiv$ can be viewed as abbreviations.

We use the following notational conventions:

$\Gamma, \Delta, \Theta, \Omega$	contexts
$\sigma, \delta, \nu, \rho$	substitutions
A, B, C	types
u, v, w, t, a, b, c	terms

In the meta level notation we omit substitutions and use variable names for referring to parts of the context.

2.1.3 Rules for context formation and substitution

The category of contexts (see section 2.2):

$$\frac{\Gamma \vdash}{1 : \Gamma \to \Gamma}\ \text{id} \qquad \frac{\sigma : \Delta \to \Gamma \quad \delta : \Theta \to \Delta}{\sigma\delta : \Theta \to \Gamma}\ \text{composition}$$

$$1\sigma \equiv \sigma \qquad \sigma 1 \equiv \sigma \qquad (\sigma\delta)\nu \equiv \sigma(\delta\nu)$$

Formation of new contexts:

$$\frac{}{() \vdash}\ \text{empty} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma.A \vdash}\ \text{comprehension}$$

Formation of new substitutions and terms:

$$\frac{\Gamma \vdash A}{\mathsf{p} : \Gamma.A \to \Gamma} \qquad \frac{\Gamma \vdash A}{\Gamma.A \vdash \mathsf{q} : A\mathsf{p}} \qquad \frac{\sigma : \Delta \to \Gamma \quad \Gamma \vdash A \quad \Delta \vdash u : A\sigma}{(\sigma, u) : \Delta \to \Gamma.A}$$

$$\mathsf{p}(\sigma, u) \equiv \sigma \qquad \mathsf{q}(\sigma, u) \equiv u \qquad 1 \equiv (\mathsf{p}, \mathsf{q}) \qquad (\sigma, u)\delta \equiv (\sigma\delta, u\delta)$$

Substitution of types and terms:

$$\frac{\Gamma \vdash A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash A\sigma} \qquad \frac{\Gamma \vdash t : A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash t\sigma : A\sigma}$$

$$(A\sigma)\delta \equiv A(\sigma\delta) \qquad A1 \equiv A \qquad (a\sigma)\delta \equiv a(\sigma\delta) \qquad a1 \equiv a$$

We will use the notation $[u] \equiv (1, u)$, $[u, v] \equiv ((1, u), v)$ etc. With this notation, if we have a type family B indexed over a type A expressed as $\Gamma.A \vdash B$, we can write $\Gamma \vdash B[a]$ for the member of the family at index a if $\Gamma \vdash a : A$. Similarly for a family indexed over two types which might depend on each other: if $\Gamma.A.B \vdash C$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B[a]$, then $\Gamma \vdash C[a, b]$.

We write p^n instead of composing p n times to shorten the notation for weakening substitutions. E.g.p^4 stands for pppp.

2.1.4 Congruence rules for definitional equality

Definitional equality is an equivalence relation, hence we have rules for expressing the reflexivity, symmetry and transitivity of $\cdot \equiv \cdot$, for contexts, substitutions, types and terms, respectively. We won't spell these out because they are completely straightforward.

We also need rules expressing that definitionally equal contexts, substitutions, types and terms can be replaced in all situations. The following coercion rules allow equal contexts and types to be replaced with each other in any judgment.

$$\frac{\Gamma \equiv \Delta \quad \Gamma \vdash A}{\Delta \vdash A} \qquad \frac{\Gamma \equiv \Delta \quad \Theta \equiv \Omega \quad \sigma : \Gamma \to \Theta}{\sigma : \Delta \to \Omega}$$

$$\frac{\Gamma \equiv \Delta \quad \Gamma \vdash A \equiv B \quad \Gamma \vdash t : A}{\Delta \vdash t : B}$$

We need additional congruence rules which express that the different ways of forming contexts, substitutions, types and terms all respect definitional equality.

For contexts, we have the following congruence rule:

$$\frac{\Gamma \equiv \Delta \quad \Gamma \vdash A \equiv B \quad \Gamma \vdash A}{\Gamma.A \equiv \Delta.B}$$

Congruence rules for substitutions:

$$\frac{\sigma \equiv \nu \quad \delta \equiv \rho}{\sigma\delta \equiv \nu\rho} \qquad \frac{\sigma \equiv \delta \quad \Gamma \vdash u \equiv v : A\sigma}{(\sigma, u) \equiv (\delta, v)}$$

Congruence rule for types:

$$\frac{\Gamma : A \equiv B \quad \sigma : \Delta \to \Gamma \quad \sigma \equiv \delta}{\Delta \vdash A\sigma \equiv B\delta}$$

Congruence rule for terms:

$$\frac{\Gamma \vdash u \equiv v : A \quad \sigma : \Delta \to \Gamma \quad \sigma \equiv \delta}{\Delta \vdash u\sigma \equiv v\delta : A\sigma}$$

Note that these rules depend on each other, e.g. in the last rule, the judgment $\Delta \vdash u\sigma \equiv v\delta : A\sigma$ does not seem to be valid, since $u\sigma$ and $u\delta$ have different types, however, by the congruence rule for types, we know that the type of $u\delta$ is $A\delta \equiv A\sigma$.

When introducing new type and term formers (as for function types in the next section), they should come with rules expressing their behaviour with regards definitional equality.

2.1.5 Rules for the function type

Rules in type theory appear following a common pattern: type formation rule, introduction rule, elimination rule, computation (β) and uniqueness (η) rules. The formation and introduction rules provide constructors for the type and terms of that type. The elimination rule provides an eliminator or destructor. Constructors and eliminators need to obey additional congruence rules expressing that definitionally equal types and terms can be replaced in every position.

In our presentation with explicit substitutions, we need to express how substitutions interact with term and type formers as well.

The function type represents implication, and more generally, universal quantification.

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B}{\Gamma \vdash \Pi\, A\, B}\ \Pi\text{-form}$$

$$\frac{\Gamma.A \vdash b : B}{\Gamma \vdash \lambda b : \Pi\, A\, B}\ \Pi\text{-intro} \qquad \frac{\Gamma \vdash w : \Pi\, A\, B \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{app}(w, a) : B[a]}\ \Pi\text{-elim}$$

Interaction with substitutions (the ξ rule below also falls into this category):

$$(\Pi\, A\, B)\sigma \equiv \Pi\, A\sigma\, B(\sigma\mathsf{p}, \mathsf{q}) \qquad \mathsf{app}(w, u)\delta \equiv \mathsf{app}(w\delta, u\delta)$$

$\sigma : \Delta \to \Gamma$, but $\Gamma.A \vdash B$, this is why we need to use the substitution $(\sigma\mathsf{p}, \mathsf{q})$, where $\sigma\mathsf{p} : \Delta.A\sigma \to \Gamma$, hence $(\sigma\mathsf{p}, \mathsf{q}) : \Delta.A\sigma \to \Gamma.A$.

We express the β (computation) rule as follows, we call it $\beta\sigma$:

$$\mathsf{app}((\lambda b)\sigma, u) \equiv b(\sigma, u)$$

The congruence rules are the following (omitting the details of contexts, types):

$$\frac{A \equiv A' \quad B \equiv B'}{\Pi\, A\, B \equiv \Pi\, A'\, B'} \qquad \frac{t \equiv t'}{\lambda t \equiv \lambda t'}$$

This defines a weak Π. If we additionally have the following rules, we get a strong Π:

$$\frac{\Gamma.A \vdash t : B}{(\lambda t)\sigma \equiv \lambda(t(\sigma\mathsf{p}, \mathsf{q}))}\ \xi \qquad \frac{\Gamma \vdash t : \Pi\, A\, B}{\Gamma \vdash t \equiv \lambda(\mathsf{app}(t\mathsf{p}, \mathsf{q}))}\ \eta\ \text{(uniqueness)}$$

The ξ rule expresses substitution under λ and it says that the indices should be shifted by one under the binder.

In a strong theory, the usual β rule $\mathsf{app}(\lambda b, u) \equiv b(1, u)$ is equivalent to $\beta\sigma$. β can be derived from $\beta\sigma$ by choosing $\sigma :\equiv 1$, the other direction follows by equational reasoning:

$$\begin{array}{ll}
\quad \mathsf{app}((\lambda b)\sigma, u) & \\
\equiv \mathsf{app}(\lambda(b(\sigma\mathsf{p}, \mathsf{q})), u) & \xi \\
\equiv b(\sigma\mathsf{p}, \mathsf{q})(1, u) & \beta \\
\equiv b(\sigma\mathsf{p}(1, u), \mathsf{q}(1, u)) & \\
\equiv b(\sigma, u) &
\end{array}$$

In what follows, we assume a weak type theory.

We use the abbreviation $A \to B$ for the type $\Pi\, A\, B\mathsf{p}$. $\to$ is right-associative, that is, $A \to B \to C$ means $A \to (B \to C)$. The scope of a λ extends as much as possible to the right hand side, eg $\lambda\lambda\mathsf{app}(w, \lambda v)$ means $\lambda(\lambda(\mathsf{app}(w, (\lambda v))))$. We abbreviate nested applications like this: $\mathsf{app}(f, x, y)$ means $\mathsf{app}(\mathsf{app}(f, x), y)$. For infix operators such as $\cdot + \cdot : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ we write $x + y$ instead of $\mathsf{app}(\cdot + \cdot, x, y)$. The dots show the positions of the arguments. For conciseness sometimes we omit some arguments of a function if they are clear from the context. We mark such implicit arguments by putting the type of the argument in subscript such as in $\mathsf{id} : \Pi_{\mathsf{U}}\, (\mathsf{El}(\mathsf{q}) \to \mathsf{El}(\mathsf{q}))$. When defining such a function, we use subscript lambdas: $\mathsf{id} :\equiv {}_{\lambda}\lambda\mathsf{q}$. When calling the function, the first parameter (which in our case has type U) can be omitted: $\mathsf{app}(\mathsf{id}, a) : \mathsf{El}(\hat{A})$ if $a : \mathsf{El}(\hat{A})$. If we want to specify it nevertheless, we give it in subscript: $\mathsf{app}(\mathsf{id},_{\hat{A}}) : \mathsf{El}(\hat{A}) \to \mathsf{El}(\hat{A})$. The meaning of the id function will become clear in section 2.1.8, for now it is enough to know that U is a type and $\mathsf{El}(t)$ is a type given $t : \mathsf{U}$. Implicit arguments only make a difference when using app to apply the function, e.g. the subscript lambdas are still referrable by substitutions etc.

Π is a binder, in $\Pi\, A\, B$, an argument of type A is bound and we can refer to it by q inside B. When using nested bindings, qp^n refers to the n^{th} binder (counting starts with 0). For example, in $\Pi\, A\, \big(\Pi\, B\, (\Pi\, C\, D(\mathsf{qp}))\big)$, qp refers to something of type B being bound by the second Π, while in $\Pi\, A\, \big(\Pi\, B\, (C \to D(\mathsf{qp}))\big)$, qp refers to the A being bound (because $\to$ is not a binder, $C \to D(\mathsf{qp})$ means $\Pi\, C\, (D(\mathsf{qp})\mathsf{p})$[4]).

The meta level notation for function types is $(a : A) \to (B[a])$ instead of $\Pi\, A\, B$, $\lambda x.\lambda y.x$ instead of $\lambda\lambda\mathsf{qp}$ and $f(a)$ instead of $\mathsf{app}(f, a)$. Sometimes we use notation with variable names for readability like in $\prod\limits_{a:A} B[a]$.

2.1.6 Rules for propositional equality

We introduce the type expressing equality inside the theory. It is called propositional equality to distinguish from the definitional equality. Definitional equality is the part of equality which can be decided (see section 2.1.12), so we build it into the system as a judgment type and rely on it when we are reasoning inside the system. We can't express definitional equality within the theory: if we do mathematics in type theory, definitional equality is a "premathematical" notion. Propositional equality is the equality which needs a proof done (programmed) by hand. The type expressing propositional equality is also called the identity type.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash u : A \quad \Gamma \vdash v : A}{\Gamma \vdash u =_A v} = \text{-form} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : a =_A a} = \text{-intro}$$

$$\frac{\Gamma.A.A\mathsf{p}.\mathsf{qp} =_{A\mathsf{pp}} \mathsf{q} \vdash C \quad \Gamma.A \vdash t : C[\mathsf{q}, \mathsf{refl}_{\mathsf{q}}] \quad \Gamma \vdash u, v : A \quad \Gamma \vdash p : u =_A v}{\Gamma \vdash \mathsf{J}(t, u, v, p) : C[u, v, p]} = \text{-elim}$$

$$\frac{\Gamma.A.A\mathsf{p}.\mathsf{qp} =_{A\mathsf{pp}} \mathsf{q} \vdash C \quad \Gamma.A \vdash t : C[\mathsf{q}, \mathsf{refl}_{\mathsf{q}}] \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{J}(t, a, a, \mathsf{refl}_a) \equiv t[a] : C[a, a, \mathsf{refl}_a]} = \text{-}\beta$$

Without explicit substitutions, the context $\Gamma.A.A\mathsf{p}.\mathsf{qp} =_{A\mathsf{pp}} \mathsf{q}$ is written as $\Gamma, x : A, y : A, z : x =_A y$.

$$(u =_A v)\sigma \equiv (u\sigma =_{A\sigma} v\sigma) \qquad (\mathsf{refl}_a)\sigma \equiv \mathsf{refl}_{a\sigma}$$

$$(\mathsf{J}(t, u, v, p))\sigma \equiv \mathsf{J}(t(\sigma\mathsf{p}, \mathsf{q}), u\sigma, v\sigma, p\sigma)$$

We have the usual congruence rules stating that if $A \equiv A'$, $u \equiv u'$ and $v \equiv v'$ then $(u =_A v) \equiv (u' =_{A'} v')$ etc.

The introduction rule is called refl because it expresses reflexivity, but also because it reflects the definitional equality inside the theory by the following

[4]This requires that D has the substitution law $D(a)\sigma \equiv D(a\sigma)$.

rule (one of the congruence rules):

$$\frac{u \equiv v : A}{\mathsf{refl}_u : u =_A v}$$

We have the following metatheoretic results concerning the relationship between definitional and propositional equality:

- For any $\Gamma \vdash a : A$, $\Gamma \vdash b : A$ we can derive the judgement $\Gamma \vdash \mathsf{refl} : a = b$ iff we can derive the judgement $\Gamma \vdash a \equiv b : A$ (the right to left direction follows from the congruence rule mentioned above, the other from $=$ -intro).
- For any $() \vdash a : A$, $() \vdash b : A$ we can derive a judgement $() \vdash p : a = b$ iff we can derive the judgement $() \vdash a \equiv b$ (the right to left direction follows from congruence and $=$ -intro, the other from canonicity: all terms in the empty context are definitionally equal to a constructor and in this case, refl is the only constructor, for details, see section 2.1.12).

It can be proved that $=_A$ is an equivalence relation for all A types i.e. given a type $\Gamma \vdash A$ we can define the following functions:

$$\begin{aligned}
&\mathsf{refl} : \Pi\, A\, (\mathsf{q} = \mathsf{q})\\
&\mathsf{refl} :\equiv \lambda \mathsf{refl}_{\mathsf{q}}\\
&\cdot^{-1} : \Pi_{\ A} \big(\Pi_{\ A}\, (\mathsf{qp} = \mathsf{q} \to \mathsf{q} = \mathsf{qp})\big)\\
&\cdot^{-1} :\equiv {}_{\lambda\,\lambda}\, \lambda \mathsf{J}(\mathsf{refl}_{\mathsf{q}}, \mathsf{qpp}, \mathsf{qp}, \mathsf{q})\\
&\cdot \blacksquare \cdot : \Pi_{\ A} \Big(\Pi_{\ A}\, \big(\mathsf{qp} = \mathsf{q} \to \Pi_{\ A}\, (\mathsf{qp} = \mathsf{q} \to \mathsf{qpp} = \mathsf{q})\big)\Big)\\
&\cdot \blacksquare \cdot :\equiv {}_{\lambda\,\lambda}\, \lambda\, {}_{\lambda}\, \lambda \mathsf{app}\Big(\mathsf{J}\big(\mathsf{J}(\mathsf{refl}, \mathsf{qp}^4, \mathsf{qp}, \mathsf{q}), \mathsf{qp}^4, \mathsf{qp}^3, \mathsf{qp}^2\big), \mathsf{qp}, \mathsf{q}\Big)\\
&\cdot \blacksquare \cdot :\equiv {}_{\lambda x.\ \lambda y.}\, \lambda p : x = y.\, {}_{\lambda z.}\, \lambda q : y = z.\mathsf{app}\Big(\mathsf{J}\big(\mathsf{J}(\mathsf{refl}, x, z, q), x, y, p\big), z, q\Big)
\end{aligned}$$

I repeated the last definition with variable names for readability.

Given a type family $\Gamma.A \vdash P$, we can transport elements of this family along equalities of A:

$$\begin{aligned}
&\mathsf{transport}^P(\cdot,\cdot) : \Pi_{\ A} \Big(\Pi_{\ A}\, \big(\mathsf{qp} = \mathsf{q} \to P[\mathsf{qp}] \to P[\mathsf{q}]\big)\Big)\\
&\mathsf{transport}^P(\cdot,\cdot) :\equiv {}_{\lambda\,\lambda}\, \lambda\lambda \mathsf{app}(\mathsf{J}(\lambda \mathsf{q}, \mathsf{qp}^3, \mathsf{qp}^2, \mathsf{qp}), \mathsf{q})
\end{aligned}$$

By using universes (section 2.1.8), these functions can be defined generically for all types in a universe (now we defined them for fixed types A and P).

2.1.7 Rules for Sigma

The rules for the sum type (also called dependent product):

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B}{\Gamma \vdash \Sigma\, A\, B}\ \Sigma\text{-form} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a]}{\Gamma \vdash (a, b) : \Sigma\, A\, B}\ \Sigma\text{-intro}$$

$$\frac{\Gamma.\Sigma\, A\, B \vdash P \quad \Gamma.A.B \vdash t : P(\mathsf{pp}, (\mathsf{qp}, \mathsf{q})) \quad \Gamma \vdash w : \Sigma\, A\, B}{\Gamma \vdash \mathsf{ind}_{\Sigma}(t, w) : P[w]}\ \Sigma\text{-elim}$$

$$\frac{}{\Gamma \vdash \mathsf{ind}_{\Sigma}(t, (a, b)) \equiv t[a, b]}\ \Sigma\text{-}\beta$$

$$(\Sigma\, A\, B)\sigma \equiv \Sigma\, A\sigma\, B(\sigma\mathsf{p}, \mathsf{q}) \qquad (u, v)\sigma \equiv (u\sigma, v\sigma)$$

$$\mathsf{ind}_\Sigma(t, w)\sigma \equiv \mathsf{ind}_\Sigma(t(\sigma\mathsf{p}, \mathsf{q}), w\sigma)$$

In the elimination rule, one can check that $(\mathsf{pp}, (\mathsf{qp}, \mathsf{q})) : \Gamma.A.B \to \Gamma.\Sigma\, A\, B$ and it means that P is substituted so that its dependency on $\Sigma\, A\, B$ is filled by the pair constructed by the last two elements in the context.

We don't assume an η rule for this type. We presented the Σ type positively. The negative presentation would provide two eliminators, the projection functions $\mathsf{pr}_1 : \Sigma\, A\, B \to A$ and $\mathsf{pr}_2 : \Pi\, (\Sigma\, A\, B)\, (B[\mathsf{pr}_1(\mathsf{q})])$ instead of ind_Σ. These can be defined inside the theory with the help of universes, see the next section. In the negative presentation, ind_Σ can only be defined if one has an (at least propositional) η rule saying that $w = (\mathsf{pr}_1(w), \mathsf{pr}_2(w))$ for all $w : \Sigma\, A\, B$. In our presentation, this (propositional) rule follows from ind_Σ.

We have the usual congruence rules, however we don't spell them out here.

The meta level notation for the sum type $\Sigma\, A\, B$ is $(a : A) \times (B[a])$. The more readable notation with variable names that we sometimes use is $\Sigma\, (a : A)\, (B[a])$ or $\sum_{a:A} B[a]$.

2.1.8 Rules for universes

A universe is a type containing other types. Universes were introduced in the first publication of type theory This can be expressed by a type U which contains codes representing types and a function El which maps codes to types:

$$\frac{}{\Gamma \vdash \mathsf{U}} \qquad \frac{\Gamma \vdash \hat{A} : \mathsf{U}}{\Gamma \vdash \mathsf{El}(\hat{A})}$$

Substitution rules:

$$\mathsf{U}\sigma \equiv \mathsf{U} \qquad (\mathsf{El}(\hat{A}))\sigma \equiv \mathsf{El}(\hat{A}\sigma)$$

I put hat on a symbol to indicate that it is a code for a type rather than a type. We need to add rules for expressing that the universe contains some types.

For example, the universe being closed under Π means that if we have a type in the universe and a type family indexed over this type, then the universe contains the dependent function space over these two types:

$$\frac{\Gamma \vdash \hat{A} : \mathsf{U} \quad \Gamma.\mathsf{El}(\hat{A}) \vdash \hat{B} : \mathsf{U}}{\Gamma \vdash \hat{\Pi}\, \hat{A}\, \hat{B} : \mathsf{U}}\ \hat{\Pi}\text{-form} \qquad \frac{\Gamma \vdash \hat{A} : \mathsf{U} \quad \Gamma.\mathsf{El}(\hat{A}) \vdash b : \mathsf{El}(\hat{B})}{\Gamma \vdash \lambda : \mathsf{El}(\hat{\Pi}\, \hat{A}\, \hat{B})}\ \hat{\Pi}\text{-intro}$$

$$\frac{\Gamma.\mathsf{El}(\hat{A}) \vdash \hat{B} : \mathsf{U} \quad \Gamma \vdash w : \mathsf{El}(\hat{\Pi}\, \hat{A}\, \hat{B}) \quad \Gamma \vdash a : \mathsf{El}(\hat{A})}{\Gamma \vdash \mathsf{app}(w, a) : \mathsf{El}(\hat{B}[a])}\ \hat{\Pi}\text{-elim}$$

We have overloaded the λ constructor and app destructor names.

The computation, substitution and congruence rules are the same as for normal Π types. Also, if we remove the "El"s, hats and ": U"s from the above three rules, we get the exact same rules as for Π. What is the difference between a universe closed under Π and having a Π type as defined in section 2.1.5? $\hat{\Pi}$ is restricted in that one is only able to form functions between types residing in the universe, while the more general Π can be used to construct functions between types no matter which universe they inhabit. However, one cannot abstract over general Π types inside the theory (only metatheoretically), but it is possible to do so with $\hat{\Pi}$ types by the following construction: if $c : \Sigma\,\mathsf{U}\,(\mathsf{El}(\mathsf{q}) \to \mathsf{U})$, then $\mathsf{El}(\hat{\Pi}\,(\mathsf{pr}_1 c)\,\mathsf{app}(\mathsf{pr}_2 c, \mathsf{q}))$ is the $\hat{\Pi}$ type corresponding to it.

One can present type theory without a judgment type $\Gamma \vdash A$ for types and instead use a universe, saying $\Gamma \vdash A : \mathsf{U}$ (omitting the El construct). However, what type should U itself have? The rule $\Gamma \vdash \mathsf{U} : \mathsf{U}$ would make the theory inconsistent as it is shown by the Burali-Forti paradox [26], or, using inductive types by the Russel-paradox (see section 2.1.10). The usual solution is a sequence of universes embedded into each other: $\mathsf{U}_0 : \mathsf{U}_1 : \mathsf{U}_2 : \ldots$, hence each universe (and, as a consequence, each term) has a type, so we could omit the judgment kind for types from the presentation. The presentation without El is called universes à la Russel [44], the current presentation is called universes à la Tarski. The former is considered as an informal version of the latter [54], the latter fits together better together with the categories with families model (section 3.1).

The hierarchy of universes is given as follows (where i is a natural number):

$$\mathsf{U} \equiv \mathsf{U}_0 \qquad \hat{\mathsf{U}} \equiv \hat{\mathsf{U}}_0 \qquad \hat{\mathsf{U}}_i : \mathsf{U}_{i+1} \qquad \mathsf{El}(\hat{\mathsf{U}}_i) \equiv \mathsf{U}_i$$

One could assert a cumulativity rule stating that $\hat{\mathsf{U}}_i : \mathsf{U}_j$ whenever $i < j$. We will only use this rule in section 2.3. The types corresponding to codes in U_0 are called *small types*.

We write U_j for universe-polymorphic definitions if we want to stress polymorphism. This means that the definition should be valid for any $j : \mathbb{N}$. Most of the time when we simply write U, it would also be a valid polymorphic definition, and we exploit this fact multiple times in section 2.3.

Universes are important for abstraction. E.g. the polymorphic identity function (defined at the end of section 2.1.5 as an example) cannot be defined without a universe. Using universes, we can define the projection functions for all Σ types as follows:

$$\begin{aligned}
&\mathsf{pr}_1\cdot : \Pi_{\,\mathsf{U}}\,(\Pi_{\,\mathsf{El}(\mathsf{q})\to\mathsf{U}}\,(\Sigma\,\mathsf{El}(\mathsf{qp})\,\mathsf{El}(\mathsf{app}(\mathsf{qp},\mathsf{q})) \to \mathsf{El}(\mathsf{qp})))\\
&\mathsf{pr}_1\cdot :\equiv {}_{\lambda}\,{}_{\lambda}\lambda\mathsf{ind}_{\Sigma}(\mathsf{qp},\mathsf{q})\\
&\mathsf{pr}_2\cdot : \Pi_{\,\mathsf{U}}\,(\Pi_{\,\mathsf{El}(\mathsf{q})\to\mathsf{U}}\,\Pi\,(\Sigma\,\mathsf{El}(\mathsf{qp})\,\mathsf{El}(\mathsf{app}(\mathsf{qp},\mathsf{q})))\,(\mathsf{El}(\mathsf{app}(\mathsf{qp},\mathsf{pr}_1\mathsf{q}))))\\
&\mathsf{pr}_2\cdot :\equiv {}_{\lambda}\,{}_{\lambda}\lambda\mathsf{ind}_{\Sigma}(\mathsf{q},\mathsf{q})
\end{aligned}$$

The expected definitional equalities $\mathsf{pr}_1(a,b) \equiv \mathsf{qp}[a,b] \equiv a$ and $\mathsf{pr}_2(a,b) \equiv \mathsf{q}[a,b] \equiv b$ hold by the β rule of Σ.

Universes are the way for defining type-level functions (functions that output a type) by defining a function which outputs a code. The above projection functions are such examples.

The universe U being closed under Σ means the following (and the usual computation and congruence rules):

$$\frac{\Gamma \vdash \hat{A} : \mathsf{U} \qquad \Gamma.\mathsf{El}(\hat{A}) \vdash \hat{B} : \mathsf{U}}{\Gamma \vdash \hat{\Sigma}\,\hat{A}\,\hat{B} : \mathsf{U}}\;\hat{\Sigma}\text{-form} \qquad \frac{\Gamma \vdash a : \mathsf{El}(\hat{A}) \qquad \Gamma \vdash b : \mathsf{El}(\hat{B}[a])}{\Gamma \vdash (a,b) : \mathsf{El}(\hat{\Sigma}\,\hat{A}\,\hat{B})}\;\hat{\Sigma}\text{-intro}$$

$$\frac{\begin{array}{l}\Gamma \vdash P : \mathsf{El}(\hat{\Sigma}\,\hat{A}\,\hat{B}) \to \mathsf{U}_j\\ \Gamma \vdash f : \Pi\,\mathsf{El}(\hat{A})\,(\Pi\,\mathsf{El}(\hat{B}[\mathsf{q}])\,\mathsf{El}(\mathsf{app}(P,(\mathsf{qp},\mathsf{q}))))\\ \Gamma \vdash w : \mathsf{El}(\hat{\Sigma}\,\hat{A}\,\hat{B})\end{array}}{\Gamma \vdash \mathsf{ind}_{\hat{\Sigma}}(P,f,w) : \mathsf{El}(\mathsf{app}(P,w))}\;\hat{\Sigma}\text{-elim}$$

If the target type of P in the elimination rule is different from U (more precisely, the index j in U_j is larger than the index of U in the hierarchy of universes), $\hat{2}$ has so-called *large elimination* (small elimination is when the codomain of P is a small type). Large elimination is required for computing a small type from a pair: we want $\mathsf{app}(f, a, b) : \mathsf{U}_0$, hence $\mathsf{El}(\mathsf{app}(P, (\mathsf{qp}, \mathsf{q}))) \equiv \mathsf{U}_0$, hence $\mathsf{app}(P, (\mathsf{qp}, \mathsf{q})) \equiv \hat{\mathsf{U}}_0 : \mathsf{U}_1$, so $j = 1$.

2.1.9 Rules for finite types

The type having no inhabitants (also called bottom, $\bot$):

$$\frac{}{\Gamma \vdash 0}\ \text{0-form} \qquad \frac{\Gamma \vdash t : 0 \quad \Gamma.0 \vdash A[t]}{\Gamma \vdash \mathsf{ind}_0(t) : A}\ \text{0-elim}$$

There is no computation rule for 0.

The type having one inhabitant (also called unit, $\top$):

$$\frac{}{\Gamma \vdash 1}\ \text{1-form} \qquad \frac{}{\Gamma \vdash * : 1}\ \text{1-intro} \qquad \frac{\Gamma.1 \vdash A \quad \Gamma \vdash u : A[*] \quad \Gamma \vdash t : 1}{\Gamma \vdash \mathsf{ind}_1(u, t) : A[t]}\ \text{1-elim}$$

$$\frac{}{\mathsf{ind}_1(t, *) \equiv t}\ \text{1-}\beta$$

The type of booleans:

$$\frac{}{\Gamma \vdash 2}\ \text{2-form} \qquad \frac{}{\Gamma \vdash \mathsf{0} : 2}\ \text{2-intro}_1 \qquad \frac{}{\Gamma \vdash \mathsf{1} : 2}\ \text{2-intro}_2$$

$$\frac{\Gamma.2 \vdash A \quad \Gamma \vdash u : A[\mathsf{0}] \quad \Gamma \vdash v : A[\mathsf{1}] \quad \Gamma \vdash t : 2}{\Gamma \vdash \mathsf{case}(u, v, t) : A[t]}\ \text{2-elim}$$

$$\frac{}{\mathsf{case}(u, v, \mathsf{0}) \equiv u}\ \text{2-}\beta_1 \qquad \frac{}{\mathsf{case}(u, v, \mathsf{1}) \equiv v}\ \text{2-}\beta_2$$

The substitution rules are the following:

$$0\sigma \equiv 0 \qquad 1\sigma \equiv 1 \qquad 2\sigma \equiv 2$$

$$*\sigma \equiv * \qquad \mathsf{1}\sigma \equiv \mathsf{1} \qquad \mathsf{2}\sigma \equiv \mathsf{2} \qquad \mathsf{case}(u, v, t)\sigma \equiv \mathsf{case}(u\sigma, v\sigma, t\sigma)$$

The congruence rules are as usual.

A universe U closed under 2 means the following:

$$\Gamma \vdash \hat{2} : \mathsf{U} \qquad \Gamma \vdash \mathsf{0} : \mathsf{El}(\hat{2}) \qquad \Gamma \vdash \mathsf{1} : \mathsf{El}(\hat{2})$$

$$\frac{\Gamma \vdash P : \mathsf{El}(\hat{2}) \to \mathsf{U}_j \quad \Gamma \vdash u : \mathsf{El}(\mathsf{app}(P, \mathsf{0})) \quad \Gamma \vdash v : \mathsf{El}(\mathsf{app}(P, \mathsf{1})) \quad \Gamma \vdash t : \mathsf{El}(\hat{2})}{\Gamma \vdash \mathsf{ind}_{\hat{2}}(P, u, v, t) : \mathsf{El}(\mathsf{app}(P, t))}$$

$$\mathsf{ind}_{\hat{2}}(P, u, v, \mathsf{0}) \equiv u \qquad \mathsf{ind}_{\hat{2}}(P, u, v, \mathsf{1}) \equiv v$$

$$\hat{2}\sigma \equiv \hat{2} \qquad \mathsf{0}\sigma \equiv \mathsf{0} \qquad \mathsf{1}\sigma \equiv \mathsf{1} \qquad \mathsf{ind}_{\hat{2}}(P, u, v, t)\sigma \equiv \mathsf{ind}_{\hat{2}}(P\sigma, u\sigma, v\sigma, t\sigma)$$

$\mathsf{ind}_{\hat{2}}$ allows large elimination: if $j = 1$, we can choose choose P to be the constant function which returns $\hat{\mathsf{U}}$ i.e. $P :\equiv \lambda\hat{\mathsf{U}} : \mathsf{El}(\hat{2}) \to \mathsf{U}_1$, and we can create a type by case splitting on a boolean: $\mathsf{ind}_{\hat{2}}(\lambda\hat{\mathsf{U}}, \hat{0}, \hat{1}, t)$ is the code for the 0-element type if $t \equiv \mathsf{0}$, otherwise it is the code for the 1-element type.

With the help of a universe closed under Π, Σ and 2 (where $\hat{2}$ has large elimination), one can define the nondependent product and sum types as $\mathsf{U} \to \mathsf{U} \to \mathsf{U}$ functions. I give some explanations for the definitions in the right column.

$\cdot\hat{\times}\cdot : \mathsf{U} \to \mathsf{U} \to \mathsf{U}$	nondependent product
$\cdot\hat{\times}\cdot :\equiv \lambda\lambda\hat{\Pi}\,\hat{2}$	$\hat{A}\hat{\times}\hat{B} \equiv \hat{\Pi}\,(t : \mathsf{El}(\hat{2}))$
$\qquad\mathsf{ind}_{\hat{2}}(\lambda\mathsf{U}, \mathsf{qpp}, \mathsf{qp}, \mathsf{q})$	$\qquad\mathsf{ind}_{\hat{2}}(\lambda x.\mathsf{U}, \hat{A}, \hat{B}, t)$
$(\cdot,\cdot) : \Pi\,{}_{\mathsf{U}}\,(\Pi\,{}_{\mathsf{U}}\,(\mathsf{El}(\mathsf{qp}) \to \mathsf{El}(\mathsf{q}) \to \mathsf{El}(\mathsf{qp}\hat{\times}\mathsf{q})))$	pairing
$(\cdot,\cdot) :\equiv {}_{\lambda}\,{}_{\lambda}\lambda$	$(\cdot,\cdot) \equiv {}_{\lambda\hat{A}:\mathsf{U}.\ \lambda\hat{B}:\mathsf{U}.}\lambda a : \mathsf{El}(\hat{A}).$
$\quad\lambda\lambda\mathsf{ind}_{\hat{2}}($	$\quad\lambda b : \mathsf{El}(\hat{B}).\lambda x : \mathsf{El}(\hat{2}).\mathsf{ind}_{\hat{2}}($
$\quad\lambda\mathsf{ind}_{\hat{2}}(\lambda\mathsf{U}, \mathsf{qp}^5, \mathsf{qp}^4, \mathsf{q}), \mathsf{qpp}, \mathsf{qp}, \mathsf{q})$	$\quad\lambda y.\mathsf{ind}_{\hat{2}}(\lambda z.\mathsf{U}, \hat{A}, \hat{B}, y), a, b, x)$
$\mathsf{pr}_1\cdot : \Pi\,{}_{\mathsf{U}}\,(\Pi\,{}_{\mathsf{U}}\,(\mathsf{El}(\mathsf{qp}\hat{\times}\mathsf{p}) \to \mathsf{El}(\mathsf{qp})))$	first projection
$\mathsf{pr}_1\cdot :\equiv {}_{\lambda}\,{}_{\lambda}\lambda\mathsf{app}(\mathsf{q}, 0)$	$\mathsf{pr}_1 w \equiv \mathsf{app}(w, 0)$
$\mathsf{pr}_2\cdot : \Pi\,{}_{\mathsf{U}}\,(\Pi\,{}_{\mathsf{U}}\,(\mathsf{El}(\mathsf{qp}\hat{\times}\mathsf{p}) \to \mathsf{El}(\mathsf{p})))$	second projection
$\mathsf{pr}_2\cdot :\equiv {}_{\lambda}\,{}_{\lambda}\lambda\mathsf{app}(\mathsf{q}, 1)$	$\mathsf{pr}_2 w \equiv \mathsf{app}(w, 1)$

We used large elimination for $\hat{2}$ in the definition of $\cdot\hat{\times}\cdot$ and in the inner usage of $\mathsf{ind}_{\hat{2}}$ in the definition of $(\cdot,\cdot)$. The β rules for the projections hold definitionally, in the first case this is $\mathsf{pr}_1(a, b) \equiv \mathsf{app}((a, b), 1) \equiv \mathsf{app}(\lambda\mathsf{ind}_{\hat{2}}(\dots, a, b, \mathsf{q}), 1) \equiv \mathsf{ind}_{\hat{2}}(\dots, a, b, 1) \equiv a$. A dependent eliminator similar to that of Σ can also be defined however only with a propositional computation rule (and this computation rule requires function extensionality, see section 2.1.13). $\cdot\hat{\times}\cdot$ can be also defined just as a special case of $\hat{\Sigma}$ by $\cdot\hat{\times}\cdot :\equiv \lambda\lambda\hat{\Sigma}\ \mathsf{qp}\ \lambda\mathsf{qp}$, in this case it has definitional or propositional computation rules just as $\hat{\Sigma}$.

$$\begin{array}{ll}
\cdot \hat{+} \cdot : \mathsf{U} \to \mathsf{U} \to \mathsf{U} & \text{nondependent sum} \\
\cdot \hat{+} \cdot :\equiv \lambda\lambda\hat{\Sigma}\,\hat{2}\,\mathsf{ind}_{\hat{2}}(\lambda\mathsf{U}, \mathsf{qpp}, \mathsf{qp}, \mathsf{q}) & \hat{A}\hat{+}\hat{B} \equiv \hat{\Sigma}\,(t : \mathsf{El}(\hat{2}))\,\mathsf{ind}_{\hat{2}}(\lambda x.\mathsf{U}\hat{A}, \hat{B}, t) \\
\mathsf{inl}\cdot : \Pi_{\,\mathsf{U}}\,(\Pi_{\,\mathsf{U}}\,(\mathsf{El}(\mathsf{qp}) \to \mathsf{El}(\mathsf{qp}\hat{+}\mathsf{q}))) & \text{first injection} \\
\mathsf{inl}\cdot :\equiv {}_{\lambda\,\lambda}\lambda(0, \mathsf{q}) & \mathsf{inl}\,u \equiv (0, u) \\
\mathsf{inr}\cdot : \Pi_{\,\mathsf{U}}\,(\Pi_{\,\mathsf{U}}\,(\mathsf{El}(\mathsf{q}) \to \mathsf{El}(\mathsf{qp}\hat{+}\mathsf{q}))) & \text{second injection} \\
\mathsf{inr}\cdot :\equiv {}_{\lambda\,\lambda}\lambda(1, \mathsf{q}) & \mathsf{inr}\,v \equiv (1, v) \\
\mathsf{ind}_{\hat{+}}(\cdot,\cdot,\cdot,\cdot) : & \mathsf{ind}_{\hat{+}}(\cdot,\cdot,\cdot,\cdot) : \\
\quad \Pi_{\,\mathsf{U}}\,(\Pi_{\,\mathsf{U}}\,(\Pi\,(\Pi\,\mathsf{El}(\mathsf{qp}\hat{+}\mathsf{q})\,\mathsf{U}_j) & \quad \Pi_{\,\hat{A}:\mathsf{U}}\,(\Pi_{\,\hat{B}:\mathsf{U}}\,(\Pi\,(P : \Pi\,\mathsf{El}(\hat{A}\hat{+}\hat{B})\,\mathsf{U}_j) \\
\qquad ((\Pi\,\mathsf{El}(\mathsf{qpp})\,\mathsf{El}(\mathsf{app}(\mathsf{qp}, \mathsf{inl}\,\mathsf{q}))) & \qquad ((\Pi\,(a : \mathsf{El}(\hat{A}))\,\mathsf{El}(\mathsf{app}(P, \mathsf{inl}\,a))) \\
\quad\; \to (\Pi\,\mathsf{El}(\mathsf{qp})\,\mathsf{El}(\mathsf{app}(\mathsf{qp}, \mathsf{inr}\,\mathsf{q}))) & \quad\; \to (\Pi\,(b : \mathsf{El}(\hat{B}))\,\mathsf{El}(\mathsf{app}(P, \mathsf{inr}\,b))) \\
\quad\; \to (\Pi\,\mathsf{El}(\mathsf{qpp}\hat{+}\mathsf{qp}) & \quad\; \to (\Pi\,(w : \mathsf{El}(\hat{A}\hat{+}\hat{B})) \\
\qquad\qquad \mathsf{El}(\mathsf{app}(\mathsf{qp}, \mathsf{q})))))) & \qquad\qquad \mathsf{El}(\mathsf{app}(P, w)))))) \\
\mathsf{ind}_{\hat{+}}(\cdot,\cdot,\cdot,\cdot) :\equiv {}_{\lambda\,\lambda}\lambda\lambda\lambda\lambda & \mathsf{ind}_{\hat{+}}(\cdot,\cdot,\cdot,\cdot) :\equiv {}_{\lambda\hat{A}.\,\lambda\hat{B}.}\lambda P.\lambda p_a.\lambda p_b.\lambda w. \\
\quad \mathsf{ind}_{\hat{\Sigma}}(\mathsf{qppp}, & \quad \mathsf{ind}_{\hat{\Sigma}}(P, \\
\qquad\quad \lambda\lambda\mathsf{app}(& \qquad\quad \lambda x.\lambda u.\mathsf{app}(\\
\qquad\qquad \mathsf{ind}_{\hat{2}}(\lambda\hat{\Pi}\,\mathsf{ind}_{\hat{2}}(\lambda\mathsf{U}, \mathsf{qp}^8, \mathsf{qp}^7, \mathsf{q}) & \qquad\qquad \mathsf{ind}_{\hat{2}}(\lambda y.\hat{\Pi}\,(v : \mathsf{ind}_{\hat{2}}(\lambda z.\mathsf{U}, \hat{A}, \hat{B}, y)) \\
\qquad\qquad\qquad\qquad \mathsf{app}(\mathsf{qp}^7, (\mathsf{qp}, \mathsf{q})), & \qquad\qquad\qquad\qquad \mathsf{app}(P, (y, v)), \\
\qquad\qquad\quad \mathsf{qp}^4, \mathsf{qp}^3, \mathsf{qp}), & \qquad\qquad\quad p_a, p_b, x), \\
\qquad\qquad \mathsf{q}), & \qquad\qquad u), \\
\qquad\quad \mathsf{q}) & \qquad\quad w)
\end{array}$$

We used large elimination for $\hat{2}$ in the definition of $\cdot\hat{+}\cdot$ and in the inner usage of $\mathsf{ind}_{\hat{2}}$ in the definition of $\mathsf{ind}_{\hat{+}}$. If $j \neq 0$, we need large elimination for $\hat{\Sigma}$ and for the other induction on $\hat{2}$ in the definition of $\mathsf{ind}_{\hat{+}}$. The β rule holds definitionally and a propositional η rule can be proved.

If the universe is closed under the type 1 as well, we have have all finite types: a type having exactly n elements can be defined by $\mathsf{El}(\hat{1}\hat{+}\ldots\hat{+}\hat{1})$ where $\hat{1}$ appears n times.

Given a universe closed under 0, 1, 2, Π, Σ and $\cdot = \cdot$ we can define the usual logical connectives by the propositions as types principle [35]:

$$\begin{array}{rll}
\mathsf{Prop} & :\equiv \mathsf{U} & \\
\neg\cdot & :\equiv \lambda\hat{\Pi}\,\mathsf{q}\,\hat{0} & \\
\cdot\wedge\cdot & :\equiv \cdot\hat{\times}\cdot & \\
\cdot\vee\cdot & :\equiv \cdot\hat{+}\cdot & \\
\cdot\Rightarrow\cdot & :\equiv \lambda\lambda\hat{\Pi}\,\mathsf{qp}\,\mathsf{qp} & (\lambda P : \mathsf{U}.\lambda Q : \mathsf{U}.\hat{\Pi}\,P\,(Q\mathsf{p})) \\
\forall_{(x:A)}P(x) & :\equiv \hat{\Pi}\,\hat{A}\,\hat{P} & \text{where } \mathsf{El}(\hat{A}) \vdash \hat{P} : \mathsf{U} \\
\exists_{(x:A)}P(x) & :\equiv \hat{\Sigma}\,\hat{A}\,\hat{P} & \text{where } \mathsf{El}(\hat{A}) \vdash \hat{P} : \mathsf{U} \\
a = b & :\equiv a \mathbin{\hat{=}} b &
\end{array}$$

2.1.10 Rules for inductive types

Inductive types are the most common data types used for programming: natural numbers, lists, trees etc.

Simple inductive types are defined by a type formation rule and a number of introduction rules. We illustrate these for the case of the natural numbers:

$$\frac{}{\Gamma \vdash \mathbb{N}}\ \mathbb{N}\text{-form} \qquad \frac{}{\Gamma \vdash \mathsf{0} : \mathbb{N}}\ \mathbb{N}\text{-intro}_1 \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{suc}(n) : \mathbb{N}}\ \mathbb{N}\text{-intro}_2$$

0 and suc are called constructors. 0 has no parameters, while suc has one parameter of type $\mathbb{N}$.

The elimination and computation rules can be derived automatically from the formation and introduction rules. The elimination rule requires a type family P indexed by $\mathbb{N}$ called *motive*, and an element of $P[c]$ for each constructor c (these are called *methods* according to [48]). The *target* t of the elimination is the natural number we want to investigate.

$$\frac{\Gamma.\mathbb{N} \vdash P \quad \Gamma \vdash z : P[0] \quad \Gamma \vdash s : \Pi\,(\Sigma\,\mathbb{N}\,P)\,P(\mathsf{p}, \mathsf{suc}(\mathsf{pr}_1(\mathsf{q}))) \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(z, s, t) : P[t]}\ \mathbb{N}\text{-elim}$$

The type of s looks like this with uncurried high level notation: $(n : \mathbb{N}) \to P(n) \to P(\mathsf{suc}(n))$.

$$\frac{}{\mathsf{ind}_{\mathbb{N}}(z, s, \mathsf{0}) \equiv z}\ \mathbb{N}\text{-}\beta_1 \qquad \frac{}{\mathsf{ind}_{\mathbb{N}}(z, s, \mathsf{suc}(n)) \equiv s(n, \mathsf{ind}_{\mathbb{N}}(z, s, n))}\ \mathbb{N}\text{-}\beta_1$$

Our previous definitions of the Σ, 0, 1, 2 types followed the same pattern, and also $\cdot \times \cdot$ and $\cdot + \cdot$ can be reformulated as inductive types. However, we introduced them already because we need them to make use of the following definitions.

We can formulate the definition of an inductive type generically by the notion of a *container* [2]. For this, we need the universe U_0 to be closed under 0, 1, 2, Π, Σ, $\cdot = \cdot$ (hence, we have $\hat{\times}$ and $\hat{+}$ as well). A container is given by a code $\hat{S} : \mathsf{U}$, the code for the type of shapes and a code $\hat{P} : \mathsf{El}(\hat{S}) \to \mathsf{U}$, the code for the type of positions depending on the shape. The shapes represent the constructors together with their non-recursive arguments, the positions represent the recursive arguments of the particular constructor. As examples we show how to encode the natural numbers, lists of $\hat{A}$s, binary trees (with $\hat{L}$s at the leafs and $\hat{N}$s at the nodes) as containers. We list the constructors together with their types and the container encodings.

$\mathsf{0} : \mathbb{N}$	$\hat{S} :\equiv \hat{1}\hat{+}\hat{1}$
$\mathsf{suc} : \mathbb{N} \to \mathbb{N}$	$\hat{P} :\equiv \lambda\mathsf{ind}_{\hat{+}}(\lambda\hat{\mathsf{U}}, \lambda\hat{0}, \lambda\hat{1}, \mathsf{q})$
$\mathsf{nil} : \mathsf{List}_{\hat{A}}$	$\hat{S} :\equiv \hat{1}\hat{+}\hat{A}$
$\mathsf{cons} : \mathsf{El}(\hat{A}) \to \mathsf{List}_{\hat{A}} \to \mathsf{List}_{\hat{A}}$	$\hat{P} :\equiv \lambda\mathsf{ind}_{\hat{+}}(\lambda\hat{\mathsf{U}}, \lambda\hat{0}, \lambda\hat{1}, \mathsf{q})$
$\mathsf{leaf} : \mathsf{El}(\hat{L}) \to \mathsf{Tree}_{\hat{L},\hat{N}}$	$\hat{S} :\equiv \hat{L}\hat{+}\hat{N}$
$\mathsf{node} : \mathsf{El}(\hat{N}) \to \mathsf{Tree}_{\hat{L},\hat{N}} \to \mathsf{Tree}_{\hat{L},\hat{N}} \to \mathsf{Tree}_{\hat{L},\hat{N}}$	$\hat{P} :\equiv \lambda\mathsf{ind}_{\hat{+}}(\lambda\hat{\mathsf{U}}, \lambda\hat{0}, \lambda\hat{2}, \mathsf{q})$

We needed large elimination on $\hat{+}$ for all three examples.

Inductive types given by a list of constructors can be automatically turned into their container encodings (this requires a strictly positive base functor to be calculated from the constructors, and then converting this functor to the appropriate codes, see section 2.2).

Given codes $\hat{S}$ and $\hat{P}$, the inductive type given by them is called a *W-type* and is defined as follows:

$$\frac{\Gamma \vdash \hat{S} : \mathsf{U} \quad \Gamma \vdash \hat{P} : \mathsf{El}(\hat{S}) \to \mathsf{U}}{\Gamma \vdash \mathsf{W}_{\hat{S},\hat{P}}}\ \textsf{W-form}$$

$$\frac{\Gamma \vdash s : \mathsf{El}(\hat{S}) \quad \Gamma \vdash f : \mathsf{El}(\mathsf{app}(\hat{P}, s)) \to \mathsf{W}_{\hat{S},\hat{P}}}{\Gamma \vdash \mathsf{sup}(s, f) : \mathsf{W}_{\hat{S},\hat{P}}}\ \textsf{W-intro}$$

sup is a constructor which requires a shape s (which original constructor it resembles together with its non-recursive arguments), and a function f which for each recursive occurrence, provides an element of the type itself. The elimination rule needs a function g which, given a shape s, an $f : \mathsf{El}(\mathsf{app}(\hat{P}, s)) \to \mathsf{W}_{\hat{S},\hat{P}}$ providing the recursive arguments for each position, and a function representing the inductive hypothesis (that is, $\mathsf{El}(\mathsf{app}(Q, \mathsf{app}(f, x)))$ for all x), states that $\mathsf{El}(\mathsf{app}(Q, \mathsf{sup}(s, f)))$.

$$\frac{\begin{array}{l}\Gamma \vdash Q : \mathsf{W}_{\hat{S},\hat{P}} \to \mathsf{U}_j \\ \Gamma \vdash g : \Pi\, \mathsf{El}(\hat{S}) \left(\Pi \left(\mathsf{El}(\mathsf{app}(\hat{P}, \mathsf{q})) \to \mathsf{W}_{\hat{S},\hat{P}}\right) \right. \\ \qquad\qquad \left(\left(\Pi\, \mathsf{El}(\mathsf{app}(\hat{P}, \mathsf{qp}))\, \mathsf{El}(\mathsf{app}(Q, \mathsf{app}(\mathsf{qp}, \mathsf{q})))\right) \right. \\ \qquad\qquad\quad \left.\left. \to \mathsf{El}(\mathsf{app}(Q, \mathsf{sup}(\mathsf{qp}, \mathsf{q})))\right)\right) \\ \Gamma \vdash w : \mathsf{W}_{\hat{S},\hat{P}}\end{array}}{\mathsf{ind}_{\mathsf{W}_{\hat{S},\hat{P}}}(Q, g, w) : \mathsf{El}(\mathsf{app}(Q, w))}\ \textsf{W-elim}$$

$$\frac{}{\mathsf{ind}_{\mathsf{W}_{\hat{S},\hat{P}}}(Q, g, \mathsf{sup}(s, f)) \equiv \mathsf{app}(g, s, f, \lambda \mathsf{ind}_{\mathsf{W}_{\hat{S},\hat{P}}}(Q, g, \mathsf{app}(f, \mathsf{q})))}\ \textsf{W-}\beta$$

And we have the usual substitution and congruence rules. A propositional η rule stating that given a $u : \Pi\, \mathsf{W}_{\hat{S},\hat{P}}\, \mathsf{El}(\mathsf{app}(P, \mathsf{q}))$ s.t. $\mathsf{app}(u, \mathsf{sup}(s, f)) = \mathsf{app}(g, s, f, u \circ f)$, $u = \lambda \mathsf{ind}_{\mathsf{W}_{\hat{S},\hat{P}}}(Q, g, \mathsf{q})$ can be proved with the help of function extensionality (section 2.1.13).

In the case of $\mathbb{N} :\equiv \mathsf{W}_{\hat{1}\hat{+}\hat{1}, \lambda \mathsf{ind}_{\hat{+}}(\lambda \hat{\mathsf{U}}, \lambda \hat{0}, \lambda \hat{1}, \mathsf{q})}$, the introduction rule needs an argument $s : \mathsf{El}(\hat{1}\hat{+}\hat{1})$, and in the zero case, when $s \equiv \mathsf{inl}\, *$, a function $f : \mathsf{El}(\hat{0}) \to \mathbb{N}$ (which can be given by $\mathsf{ind}_{\hat{0}}$). In the successor case, when $s \equiv \mathsf{inr}\, *$, a function $f : \mathsf{El}(\hat{1}) \to \mathbb{N}$, which is equivalent to giving the argument of type $\mathbb{N}$. However, defining the eliminator by using W-elim needs function extensionality (section 2.1.13), see [5].

The usage of elimination rules like $\mathsf{ind}_{\mathsf{W}}$ in definitions of functions can be replaced by the more user-friendly tool called *pattern matching* [29]. Instead of specifying methods as parameters of the eliminator, we specify them in separate lines for each method by putting the λ-bindings on the left hand side of the $:\equiv$ as parameters of the constructor. The left hand sides are called patterns. If we pattern match on something of type 0, we don't need to give a right hand side. For example, addition on natural numbers can be defined as follows:

$$\begin{array}{l}\mathsf{add}(\cdot, \cdot) : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \mathsf{add}(\mathsf{sup}(\mathsf{inl}\, *, f), n) :\equiv n \\ \mathsf{add}(\mathsf{sup}(\mathsf{inr}\, *, f), n) :\equiv \mathsf{sup}(\mathsf{inr}\, *, \mathsf{add}(\mathsf{app}(f, *), n))\end{array}$$

This can be translated to a normal function definition by a combination of $\mathsf{ind}_{\mathsf{W}}$ and $\mathsf{ind}_{\hat{+}}$. The translation from pattern matching to eliminators requires an additional eliminator for propositional equality called K [60]:

$$\frac{\Gamma.A.\mathsf{q} =_{A\mathsf{p}} \mathsf{q} \vdash C \quad \Gamma.A \vdash t : C[\mathsf{refl}_{\mathsf{q}}] \quad \Gamma \vdash a : A \quad \Gamma \vdash p : a =_A a}{\Gamma \vdash \mathsf{K}(t, a, p) : C[a, p]} = \text{-elim}_2$$

This eliminator is a variant of J defined in section 2.1.6: the difference is that the two parameters of $\cdot = \cdot$ are the same. It does not follow from J [34]. K implies uniqueness of identity proofs (every two elements of a type $u =_A v$ are equal). Our usages of pattern matching below can always be translated to eliminators without using K.

A universe closed under W-types means the above rules decorated with "El"s and "U"s in the appropriate places. The formation rule is:

$$\frac{\Gamma \vdash \hat{S} : \mathsf{U} \quad \Gamma \vdash \hat{P} : \mathsf{El}(\hat{S}) \to \mathsf{U}}{\Gamma \vdash \hat{\mathsf{W}}_{\hat{S},\hat{P}} : \mathsf{U}} \ \hat{\mathsf{W}}\text{-form}$$

If we had the rule $\hat{\mathsf{U}} : \mathsf{U}$, we would be able to give a W-type $\mathsf{El}(\hat{\mathsf{T}})$ representing trees having arbitrary number of branches at each node by taking $\hat{S} :\equiv \hat{U}$ which represents the set of branches for the tree (e.g. if the shape is $\hat{2} : \mathsf{El}(\hat{U})$, there are two branches of the tree), and given any $\hat{I} : \mathsf{El}(\hat{S}) \equiv \mathsf{U}$, $\mathsf{app}(\hat{P}, \hat{I}) :\equiv \hat{I}$. The introduction rule now has the following form:

$$\frac{\hat{I} : \mathsf{U} \quad f : \mathsf{El}(\hat{I}) \to \mathsf{El}(\hat{\mathsf{T}})}{\mathsf{sup}(\hat{I}, f) : \mathsf{El}(\hat{\mathsf{T}})}$$

We can think about elements of $\mathsf{El}(\hat{\mathsf{T}})$ as sets. The set $\mathsf{sup}(\hat{I}, f)$ has as many elements as the type $\mathsf{El}(\hat{I})$ and each such element is again a set given by the function f. With this definition, Russel's paradox can be encoded [16]: a set is defined as normal if it does not contain itself: $\mathsf{normal}(\mathsf{sup}(\hat{I}, f)) :\equiv \hat{\Pi}\, \hat{I}\, (\neg(\mathsf{app}(f, \mathsf{q}) \mathrel{\hat{=}} \mathsf{sup}(\hat{I}, f)))$. We can also define the set N of all normal sets: $\mathsf{N} :\equiv \mathsf{sup}(\hat{\Sigma}\, \hat{\mathsf{T}}\, \mathsf{normal}(\mathsf{q}), \mathsf{pr}_1)$. It can be shown that N is not normal: this means that from N being normal we can derive an element of $\mathsf{El}(\hat{0})$. N being normal means that given a set, a proof that it is normal and another proof that it is equal to N, we can derive $\mathsf{El}(\hat{0})$. But we can just give N itself as the set, the proof that it is normal as the proof that it is normal and $\mathsf{refl}_{\mathsf{N}}$ as the proof that it is equal to N. On a higher level this proof says that if N is normal, it has to contain itself, because it is just defined as the set of all normal sets, but then we arrive to a contradiction:

$$\begin{aligned}
&\mathsf{nonnorm}_{\mathsf{N}} : \mathsf{El}(\mathsf{normal}(\mathsf{N}) \mathrel{\hat{\to}} \hat{0}) \equiv \mathsf{El}(\hat{\Pi}\, (\hat{\Sigma}\, \hat{T}\, \mathsf{normal}(\mathsf{q}))\, (\mathsf{pr}_1\mathsf{q} = \mathsf{N} \mathrel{\hat{\to}} \hat{0}) \mathrel{\hat{\to}} \hat{0}) \\
&\mathsf{nonnorm}_{\mathsf{N}}(norm_{\mathsf{N}}) :\equiv \mathsf{app}(norm_{\mathsf{N}}, (\mathsf{N}, norm_{\mathsf{N}}), \mathsf{refl}_{\mathsf{N}})
\end{aligned}$$

We can also prove that N is normal: this means that given a set t, a proof n that it is normal and a proof p that it is equal to N, we need to derive $\mathsf{El}(\hat{0})$. n says that t is normal, but because of $p : t = \mathsf{N}$, we also know that N is normal, so we can plug in t as the set, n as the proof that it is normal and p as the proof that it is equal to N, and get $\mathsf{El}(\hat{0})$. On a higher level this says that if N contains itself, then we know that it is normal, since all of its elements are normal:

$$\begin{aligned}&\mathsf{norm}_{\mathsf{N}} : \mathsf{El}(\hat{\Pi}\,(\hat{\Sigma}\,\hat{\mathsf{T}}\,\mathsf{normal}(\mathsf{q}))\,(\mathsf{pr}_1\mathsf{q} \mathrel{\hat{=}} \mathsf{N} \mathrel{\hat{\to}} \hat{0}))\\ &\mathsf{norm}_{\mathsf{N}}((t,n),p) :\equiv \mathsf{app}(\mathsf{transport}^{\mathsf{normal}}(p,n),t,n,p)\end{aligned}$$

Simple inductive types as described above don't capture data types carrying additional information about their elements such as lists indexed by their length or data types carrying some invariant. The notions of container and W-type can be generalized to that of indexed container [52] and indexed W-type to cover these cases as well. We follow the presentation of [42].

An indexed container is given by the following data:

$\hat{I} : \mathsf{U}$	the indexing type
$\hat{S} : \mathsf{El}(\hat{I}) \to \mathsf{U}$	given an output index, the type of shapes
$\hat{P} : \Pi_{\,\mathsf{El}(\hat{I})}\,\big(\mathsf{El}(\mathsf{app}(\hat{S},\mathsf{q})) \to \mathsf{U}\big)$	given a shape, the type of positions
$r : \Pi_{\,\mathsf{El}(\hat{I})}\,\Big(\Pi_{\,\mathsf{El}(\mathsf{app}(\hat{S},\mathsf{q}))}\,\big(\mathsf{El}(\mathsf{app}(\hat{P},\,{}_{\mathsf{qp}},\mathsf{q})) \to \mathsf{El}(\hat{I})\big)\Big)$	given a position, its input index

When defining an indexed container, we define shapes by induction on the output index and positions by induction on output index and shape. The input index of a recursive argument however is given by a separate function.

As an example we give the constructors for the type of vectors (lists of A indexed by natural numbers providing their length) and the container representing this type.

$$\begin{aligned}&\mathsf{nil} : \mathsf{Vec}_{\hat{A}}(0)\\ &\mathsf{cons} : \Pi_{\,\mathbb{N}}\,\big(\mathsf{El}(\hat{A}) \to \mathsf{Vec}_{\hat{A}}(\mathsf{q}) \to \mathsf{Vec}_{\hat{A}}(\mathsf{suc}(\mathsf{q}))\big)\end{aligned}$$

$\hat{I}$	$:\equiv \hat{\mathbb{N}}$	indices are natural numbers
$\hat{S}(0)$	$:\equiv \hat{1}$	shape for the nil constructor
$\hat{S}(\mathsf{suc}(n))$	$:\equiv \hat{A}$	the cons constructor has one parameter of type $\mathbb{N}$
$\hat{P}({}_0, s)$	$:\equiv \hat{0}$	nil has no recursive arguments
$\hat{P}({}_{\mathsf{suc}(n)}, a)$	$:\equiv \hat{1}$	cons has 1 recursive argument
$r({}_0,\ {}_*, ())$		the empty pattern is denoted by ()
$r({}_{\mathsf{suc}(n)},\ {}_a, *)$	$:\equiv n$	the recursive argument of cons should have input index n if the output index is $\mathsf{suc}(n)$

Given an indexed container by $\hat{I}$, $\hat{S}$, $\hat{P}$, r, the indexed W-type corresponding to it is given by:

$$\frac{\begin{array}{l}\Gamma \vdash \hat{I} : \mathsf{U}\\ \Gamma \vdash \hat{S} : \mathsf{El}(\hat{I}) \to \mathsf{U}\\ \Gamma \vdash \hat{P} : \Pi_{\,\mathsf{El}(\hat{I})}\,\big(\mathsf{El}(\mathsf{app}(\hat{S},\mathsf{q})) \to \mathsf{U}\big)\\ \Gamma \vdash r : \Pi_{\,\mathsf{El}(\hat{I})}\,\Big(\Pi_{\,\mathsf{El}(\mathsf{app}(\hat{S},\mathsf{q}))}\,\big(\mathsf{El}(\mathsf{app}(\hat{P},\mathsf{q})) \to \mathsf{El}(\hat{I})\big)\Big)\\ \Gamma \vdash i : \mathsf{El}(\hat{I})\end{array}}{\Gamma \vdash \mathsf{W}_{\hat{I},\hat{S},\hat{P},r}(i)}\ \text{W-form}$$

$$\frac{\begin{array}{l}\Gamma \vdash i : \mathsf{El}(\hat{I}) \\ \Gamma \vdash s : \mathsf{El}(\mathsf{app}(\hat{S}, i)) \\ \Gamma \vdash f : \Pi\, \mathsf{El}(\mathsf{app}(\hat{P}, s))\; \mathsf{W}_{\hat{I},\hat{S},\hat{P},r}(\mathsf{app}(r, \mathsf{q}))\end{array}}{\mathsf{sup}(i, s, f) : \mathsf{W}_{\hat{I},\hat{S},\hat{P},r}(i)}\ \text{W-intro}$$

$$\frac{\begin{array}{l}\Gamma \vdash Q : \Pi\, \mathsf{El}(\hat{I})\; (\mathsf{W}_{\hat{I},\hat{S},\hat{P},r}(\mathsf{q}) \to \mathsf{U}_j) \\ \Gamma \vdash g : \Pi\, \mathsf{El}(\hat{I}) \Bigg(\Pi\, \mathsf{El}(\mathsf{app}(\hat{S}, \mathsf{q})) \\ \qquad\qquad \Bigg(\Pi \Big(\Pi\, \mathsf{El}(\mathsf{app}(\hat{P}, \mathsf{q}))\; \mathsf{W}_{\hat{I},\hat{S},\hat{P},r}(\mathsf{app}(r, \mathsf{q}))\Big) \\ \qquad\qquad\quad \Big(\big(\Pi\, \mathsf{El}(\mathsf{app}(\hat{P}, \mathsf{qp})) \\ \qquad\qquad\qquad \mathsf{El}(\mathsf{app}(Q, \mathsf{app}(r, \mathsf{q}), \mathsf{app}(\mathsf{qp}, \mathsf{q})))\big) \\ \qquad\qquad\qquad \to \mathsf{El}(\mathsf{app}(Q, \mathsf{qpp}, \mathsf{sup}(\mathsf{qpp}, \mathsf{qp}, \mathsf{q})))\Big)\Bigg)\Bigg) \\ \Gamma \vdash i : \mathsf{El}(\hat{I}) \\ \Gamma \vdash w : \mathsf{W}_{\hat{I},\hat{S},\hat{P},r}(i)\end{array}}{\mathsf{ind}_{\mathsf{W}_{\hat{I},\hat{S},\hat{P},r}}(Q, g, w) : \mathsf{El}(\mathsf{app}(Q, i, w))}\ \text{W-elim}$$

For helping readability, here is the type of the above g in high level notation omitting "El"s and "app"s:

$$\begin{array}{rl} g : & (i : I)\; (s : S(i)) \\ & \Big(f : \big(p : P(s)\big) \to \mathsf{W}\big(r(p)\big)\Big) \\ \to & \Big(\big(p : P(s)\big) \to Q\big(r(p), f(p)\big)\Big) \\ \to & Q(i, \mathsf{sup}(i, s, f)) \end{array}$$

The computation rule:

$$\frac{}{\mathsf{ind}_{\mathsf{W}_{\hat{S},\hat{P}}}(Q, g, \mathsf{sup}(i, s, f)) \equiv \mathsf{app}(g, i, s, f, \lambda \mathsf{ind}_{\mathsf{W}_{\hat{I},\hat{S},\hat{P},r}}(Q, g, \mathsf{app}(f, \mathsf{q})))}\ \text{W-}\beta$$

To give a proper type theoretic foundation for a programming language, one needs to provide tools for writing non-terminating programs such as servers. These are called coinductively defined coprograms to distinguish them from inductively defined programs. The coinductive versions of W-types are called M-types, their usage in type theory is summarized e.g. in [13].

2.1.11 Induction-recursion

As we would like to use type theory as our metatheory, we should be able to internalize all the above judgment types and rules inside type theory. For example we would like to internalize a universe V closed under 0, 1, + and Π. We assume that our metatheory has a universe U closed under 0, 1, 2, Π and W-types. We define V as an inductive type $\mathsf{El}(\hat{\mathsf{V}})$, the type of codes. We specify it by its formation rule, the list of its constructors and a decoding function Elem

to the universe U. We write codes in $\hat{\mathsf{V}}$ by lowercase letters.

$$\begin{aligned}
&\hat{\mathsf{V}} &&: \mathsf{U}\\
&\mathsf{zero} &&: \mathsf{El}(\hat{\mathsf{V}})\\
&\mathsf{one} &&: \mathsf{El}(\hat{\mathsf{V}})\\
&\mathsf{plus}(\cdot,\cdot) &&: \mathsf{El}(\hat{\mathsf{V}}) \to \mathsf{El}(\hat{\mathsf{V}}) \to \mathsf{El}(\hat{\mathsf{V}})\\
&\mathsf{pi}(\cdot,\cdot) &&: \Pi\ \mathsf{El}(\hat{\mathsf{V}})\ \Big(\big(\mathsf{El}(\mathsf{Elem}(\mathsf{q})) \to \mathsf{El}(\hat{\mathsf{V}})\big) \to \mathsf{El}(\hat{\mathsf{V}})\Big)
\end{aligned}$$

pi expresses the following rule, which is very similar to the rule formation rule of $\hat{\Pi}$ given in section 2.1.8.

$$\frac{\Gamma \vdash a : \mathsf{El}(\hat{\mathsf{V}}) \quad \Gamma.\mathsf{El}(\mathsf{Elem}(a)) \vdash b : \mathsf{El}(\hat{\mathsf{V}})}{\Gamma \vdash \mathsf{pi}(a,b) : \mathsf{El}(\hat{\mathsf{V}})}$$

The decoding function is defined by pattern matching as follows:

$$\begin{aligned}
&\mathsf{Elem}(\mathsf{zero}) &&:\equiv \hat{0}\\
&\mathsf{Elem}(\mathsf{one}) &&:\equiv \hat{1}\\
&\mathsf{Elem}(\mathsf{plus}(a,b)) &&:\equiv \mathsf{Elem}(a)\hat{+}\mathsf{Elem}(b)\\
&\mathsf{Elem}(\mathsf{pi}(a,b)) &&:\equiv \hat{\Pi}\ \mathsf{Elem}(a)\ \mathsf{Elem}(\mathsf{app}(b,\mathsf{q}))
\end{aligned}$$

Now for example the type

$$\mathsf{Elem}\Bigg(\mathsf{pi}\Big(\mathsf{plus}(\mathsf{one},\mathsf{one}), \mathsf{ind}_{\hat{+}}\big(\lambda\hat{\mathsf{V}}, \lambda\mathsf{one}, \lambda\mathsf{plus}(\mathsf{plus}(\mathsf{one},\mathsf{one}),\mathsf{one}), \mathsf{q}\big)\Big)\Bigg)$$

corresponds to the functions from the two element type $(\hat{1}\hat{+}\hat{1})$ to the one-element type if the input of the function was $\mathsf{inl}\,*$, and the three-element type if the input of the function was $\mathsf{inr}\,*$.

V cannot be turned directly into a W-type because what would the position corresponding to the shape be for the pi constructor? It uses the function Elem, which is in turn defined by induction on $\hat{\mathsf{V}}$. These mutual definitions of a data type and a function over that data type can be described by the theory of induction-recursion [23]. However, small inductive recursive definitions (where the target type of the function is a small type) can be converted into an indexed container [42]. The idea is to index the data type with the return type of the function defined simultaneously with it and the return types of the constructors should reflect what the function was doing: essentially we encode the function in the indices of the data type. In our case U is not a small type, but we can apply the same technique to our case defining a data type V' indexed by U residing in U_1.

$$\begin{aligned}
&\hat{\mathsf{V}}' &&: \mathsf{U} \to \mathsf{U}_1\\
&\mathsf{zero} &&: \mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}', \hat{0}))\\
&\mathsf{one} &&: \mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}', \hat{1}))\\
&\mathsf{plus}(\cdot,\cdot) &&: \Pi_{\mathsf{U}}\ \Big(\Pi_{\mathsf{U}}\ \big(\mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}',\mathsf{qp})) \to \mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}',\mathsf{q})) \to \mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}',\mathsf{qp}\hat{+}\mathsf{q}))\big)\Big)\\
&\mathsf{pi}(\cdot,\cdot) &&: \Pi_{\mathsf{U}}\ \Bigg(\Pi_{\mathsf{El}(\mathsf{q})\to\mathsf{U}}\ \Big(\mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}',\mathsf{qp})) \to \big(\Pi\ \mathsf{El}(\mathsf{qp})\ \mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}',\mathsf{app}(\mathsf{qp},\mathsf{q})))\big)\\
&&&\qquad \to \mathsf{El}(\mathsf{app}(\hat{\mathsf{V}}',\hat{\Pi}\ \mathsf{qp}\ \mathsf{app}(\mathsf{qp},\mathsf{q})))\Big)\Bigg)
\end{aligned}$$

The first implicit parameter of plus represents the output of Elem for the first non-implicit parameter, and this output is the index of the first non-implicit parameter, similarly the second implicit parameter. The type of pi using high level notation omitting “El”s:

$$\mathsf{pi} :_{(\hat{A}:\mathsf{U})(\hat{B}:\hat{A}\to\mathsf{U})} \to \hat{\mathsf{V}}'(\hat{A}) \to \big((x:\hat{A}) \to \hat{\mathsf{V}}'(\hat{B}(x))\big) \to \hat{\mathsf{V}}'\big((x:\hat{A})\hat{\to}\hat{B}(x)\big)$$

$\hat{A}$ is the type corresponding to the parameter of type $\hat{\mathsf{V}}'(\hat{A})$, hence the second parameter's type is a function from $\hat{A}$ to $\hat{\mathsf{V}}'$ parameterised with the corresponding output of Elem for the second non-implicit parameter, which depends on the first.

This definition can be encoded as an indexed W-type if we have W-types that can be indexed by large types. The relationship between the original inductive-recursive and this new type is not clear to me, and it seems that the metatheory needs to have (large) induction-recursion to be able to cope with such definitions.

2.1.12 Computation and metatheory

The way computation is expressed in type theory is normalisation. Normal forms of terms are the results of a program that we are interested in. For example, suc(suc(suc(0))) is the normal form of the natural number representing 3, while suc(0) + (suc(suc(0)) + 0) is not in normal form i.e. it needs more computation to be done to see the result. The following is the general definition of normal forms for a type theory with function types:

$$\begin{aligned} v &::= \lambda v \mid n \\ n &::= x \mid \mathsf{app}(n, v) \end{aligned}$$

where x is a variable i.e. q or qp or qpp etc. v represents values in normal form. So a value is either a lambda abstraction of a value or a neutral term, and a neutral term is either a variable or a neutral term applied to a value. Neutral terms are open terms with variables which cannot reduce any further - when an eliminator like app is applied to a neutral term it produces another neutral term. The normal forms are typed but we do not express this in our informal notation.

In the presence of other term formers, an introduction rule gives a new way of introducing a value, the elimination rule a new way of introducing a neutral term. E.g. if we have Σ types as well, these are the normal forms:

$$\begin{aligned} v &::= \lambda v \mid n \mid (v, v) \\ n &::= x \mid \mathsf{app}(n, v) \mid \mathsf{ind}_\Sigma(v, n) \end{aligned}$$

A program is expressed as a term and executing the program normalises the term, i.e. gives the normal form of the term. In order to give computational meaning to the type theory, we need to provide an algorithm which given a term computes its normal form. There are several ways to do this:

- By orienting all the definitional equalities in the formal system defined above, one gets a small-step rewrite system. We defined these equalities in sections 2.1.3 and 2.1.5 so that the orientation from left to right gives a rewrite system which corresponds to the usual definition of rewrite systems for lambda calculi. The rewriting system is *confluent* if for all t terms, if t reduces to t' and also to t'' by applying a different sequence of rewriting steps, there exists a t''' s.t. both t' and t'' reduce to it. The rewriting system has *strong normalisation*, if for any term, any sequence of rewriting steps terminates. If we have confluence and strong normalisation we can extract a definition of normal forms from the shape of those terms which do not reduce anymore. This should coincide with the definition of normal forms as given above; the method of computing normal forms is just applying (any) rewriting step until there are no more rewriting rules to be applied. However, strong normalisation for the simply typed λ-calculus with explicit subsitutions is not true [50], and this applies to our case as well.

- Big-step normalisation is an approach which evaluates programs to weak-head normal forms[5] by an environment machine and then applies this method recursively to get an actual normal form. To prove that such a normaliser terminates, the normaliser is augmented with an inductively defined reduction relation expressing the big reduction steps (Bove-Capretta technique, see [12]). Type theory with explicit substitutions together with a universe closed under Pi-types is proved normalising in [15].

- Normalisation by evaluation [11] is another technique which works by model construction (see section 3).

Logical consistency of the theory follows from normalisation: the empty type does not have any inhabitants because all terms are definitionally equal to a normal form and there is no introduction rule for the empty type, so there is no term in normal form with the empty type as its type.

If we have an algorithm to compute the normal form of any term and normal forms have decidable equality, we get that definitional equality is decidable: to decide whether two terms are definitionally equal, we just normalise them and compare the normal forms. If we have the additional property that normalisation is surjective, i.e. the collection of normal forms doesn't contain redundant elements, we can reason inductively about terms by their normal forms.

2.1.13 Extensionality

The type theory as presented above lacks some rules which are commonly used in informal mathematics. These rules are associated with how propositional equality works. One rule is function extensionality which states that two pointwise equal functions are equal:

$$\frac{\Gamma \vdash f : \Pi\, A\, B \quad \Gamma \vdash g : \Pi\, A\, B \quad \Gamma \vdash t : \Pi\, A\, \big(\mathsf{app}(f, \mathsf{q}) =_B \mathsf{app}(g, \mathsf{q})\big)}{\Gamma \vdash \mathsf{funext}(t) : f =_{\Pi\, A\, B}\, g}$$

We could assert this rule, but then we would lose normalisation: we give a new constructor for propositional equality called funext but the elimination rule J only works for the case when equality was introduced by refl. Then, every

[5]In weak-head normal form, the outermost constructor is visible, however, parameters of the outermost constructor need not be in normal form.

function defined by J will be stuck when receiving $\mathsf{funext}(...)$ as its target, so it will not reduce to a normal form as defined in section 2.1.12.

The situation is similar with another informal practice in mathematics: identifying isomorphic types. Two types A, B are isomorphic if there is a function $f : \mathsf{El}(\hat{A}) \to \mathsf{El}(\hat{B})$, a function $g : \mathsf{El}(\hat{B}) \to \mathsf{El}(\hat{A})$ s.t. for all $x : \mathsf{El}(\hat{A}) \,.\, \mathsf{app}(g, \mathsf{app}(f, x)) =_{\mathsf{El}(\hat{A})} x$ and for all $y : \mathsf{El}(\hat{B}) \,.\, \mathsf{app}(f, \mathsf{app}(g, y)) =_{\mathsf{El}(\hat{B})} y$. In mathematics, if this is the case, one can replace A and B in any situation by just transporting elements using the isomorphism. In type theory, this would correspond to an element $\mathsf{isotoid}(i) : \hat{A} =_{\mathsf{U}} \hat{B}$ where i represents the isomorphism. For example, if A has an operation $\cdot \circ \cdot : \mathsf{El}(\hat{A} \hat{\to} \hat{A} \hat{\to} \hat{A})$ we should be able to use the $\mathsf{transport}$ function to get an operation $\cdot \circ' \cdot : \mathsf{El}(\hat{B} \hat{\to} \hat{B} \hat{\to} \hat{B})$:

$$\cdot \circ' \cdot :\equiv \mathsf{transport}^{\lambda \mathsf{q} \to \mathsf{q} \to \mathsf{q}}(\mathsf{isotoid}(i), \cdot \circ \cdot)$$

However, as before, $\mathsf{transport}$ would not compute, because it is defined by J which only computes when the target is refl:

$$\mathsf{transport}^{\lambda \mathsf{q} \to \mathsf{q} \to \mathsf{q}}(\mathsf{isotoid}(i), \cdot \circ \cdot) \equiv \mathsf{app}(\mathsf{J}(\lambda \mathsf{q}, \hat{A}, \hat{B}, \mathsf{isotoid}(i)), \cdot \circ \cdot) \equiv ?$$

We can still use these rules inside our theory but we need to do all the computation involving these terms by hand. We know that these rules do not make the theory inconsistent by Voevodsky's simplicial set model [36]. For details, see section 2.3.

Another way of providing extensionality is having an equality reflection rule in our theory which says:

$$\frac{p : u =_A v}{u \equiv v : A} \; = \text{-reflection}$$

A type theory with such a rule is called *extensional type theory*. Here extensionality refers to that of the identity type, i.e. the identity type is determined by it's extensions: the collection of pairs which are equal. This is not directly induced by the above rule but can be derived: $x : A, y : A, p : x =_A y \vdash p =_{x=y} \mathsf{refl}$ can be proved by J. By having such a rule, type checking (proof checking) becomes undecidable as the well-typedness of a term can depend on the equality of two $\mathbb{N} \to \mathbb{N}$ functions which is undecidable in general. However, function extensionality can be derived as a theorem in extensional type theory:

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{refl} : \lambda x.f(x) = \lambda x.f(x) \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash p : (x : A) \to f(x) = g(x)}{\Gamma, x : A \vdash \mathsf{app}(p, x) : f(x) = g(x)} \; \mathsf{app} \text{ and weakening}}{\Gamma, x : A \vdash f(x) \equiv g(x)} \; = \text{-reflection}}{\Gamma \vdash \lambda x.f(x) \equiv \lambda x.g(x)} \; \lambda\text{-cong}}{\Gamma \vdash \mathsf{refl} : \lambda x.f(x) = \lambda x.g(x)} \; = \text{-cong}}{\Gamma \vdash \mathsf{refl} : f = g} \; \eta \text{ for functions}$$

A proof term cannot represent the above derivation fully in extensional type theory, because there is no way of representing the usage of equality reflection. The type checker must be able to reproduce the whole above derivation tree from the proof term containing little information.

2.1.14 Notes

Martin-Löf's first publication of type theory [43] was a weak theory with intensional equality and did not use contexts explicitly. Explicit substitutions first appeared in [1] as a tool to understand implementations which postpone substitutions to benefit from sharing. They are a helpful tool for presenting weak type theories as investigated by [17]. Another presentation of the syntax with explicit substitutions geared towards semantics is given in [33]. Implicit arguments were introduced by [57]. Universes were introduced in [43].

Containers were introduced in [2], indexed containers in [52], induction-recursion in [24]. Their relationship is studied in [42].

A good introduction to type theory and its relation to logic is [28]. A modern presentation of Martin-Löf's type theory can be found in [58]. It also has a section about metatheory.

Big-step normalisation is studied in [4].

The relationship between positive and negative presentations of types is studied by linear logic [27].

2.2 Category Theory

Category theory is a branch of mathematics providing a unified, high level language which can be used to talk about type theories. We briefly introduce the basic concepts of category theory which we will use later: categories, functors, natural transformations, adjunctions and we give an intuition about higher category theory.

As mathematical foundation for our following definitions we use type theory as introduced in the previous section.

2.2.1 Categories

Definition 2.1 (Set)**.** *A type T is a set, if it has unique identity proofs, that is, for all $x, y : T$, $p, q : x =_T y$, we have $p =_{x=y} q$. We use the notation $T : \mathbf{Set}$. We give a formal definition in 2.27.*

Definition 2.2 (Category)**.** *A category $\mathcal{C}$ is given by the following data:*

- *a type of objects $|\mathcal{C}|$*
- *for any two objects A, B, a set[6] of morphisms $Hom_{\mathcal{C}}(A, B)$, $f : Hom_{\mathcal{C}}(A, B)$ is also denoted as $f : A \to B$*
- *an infix composition function $\cdot \circ \cdot : Hom_{\mathcal{C}}(B, C) \to Hom_{\mathcal{C}}(A, B) \to Hom_{\mathcal{C}}(A, C)$*
- *for each object A an identity morphism $1_A : A \to A$*

For which the following laws hold:

- *composition is associative*

[6] If $Hom_{\mathcal{C}}$ was a type without such a restriction, we would need additional coherence laws apart from the given associativity and composition rules e.g. stating that the two different ways of using the associativity rule to show that the composition of 4 morphisms are equal, are equal. We will come back to this in section 2.3.

- *left and right composition with the identity morphism is the same as the original morphism*

A category can be thought of as a generalized monoid where instead of one carrier we have multiple carriers and all elements of the monoid are "typed": they have a domain and codomain; the monoid operation only works in a type-safe way: two elements can be multiplied if their codomain and domain matches.

Definition 2.3 (**1**)**.** *The one-object category* **1** *has one object* $*$ *and one identity morphism* $1_* : * \to *$.

Definition 2.4 (**Set**)**.** *The category* **Set** *is given by the following data:*

- *objects are sets (as given in definition 2.1)*
- *morphisms from* A *to* B *are functions of type* $A \to B$
- *composition is usual function composition*
- *identities are the identity functions defined by* $\lambda \mathsf{q}$

We have intentionally chosen the same name for the category **Set** and the type of sets, since the type of sets can be given the structure of a category and we use the same name for the category.

Definition 2.5 (Section)**.** *A section of* $f : A \to B$ *is a morphism* $g : B \to A$ *s.t.* $f \circ g = 1_B$ *(a right inverse of* f*).*

Definition 2.6 (Isomorphism)**.** *A morphism* $f : A \to B$ *is called an isomorphism if there is a morphism* $g : B \to A$ *s.t.* $g \circ f = 1_A$ *and* $f \circ g = 1_B$.

Definition 2.7 (Groupoid)**.** *A groupoid is a category where every morphism is an isomorphism.*

Definition 2.8 (Dual of a category)**.** *A dual of a category* $\mathcal{C}$ *is a category* $\mathcal{C}^{op}$ *defined by the following data:*

- $|\mathcal{C}^{op}| :\equiv |\mathcal{C}|$
- *for all* $A, B : |\mathcal{C}^{op}|$, $Hom_{\mathcal{C}^{op}}(A, B) :\equiv Hom_{\mathcal{C}}(B, A)$
- *if* $f : Hom_{\mathcal{C}^{op}}(A, B) \equiv Hom_{\mathcal{C}}(B, A)$, $g : Hom_{\mathcal{C}^{op}}(B, C) \equiv Hom_{\mathcal{C}}(C, B)$, $g \circ_{\mathcal{C}^{op}} f :\equiv f \circ_{\mathcal{C}} g : Hom_{\mathcal{C}}(C, A) \equiv Hom_{\mathcal{C}^{op}}(A, C)$
- *identities are the same as in* $\mathcal{C}$

Definition 2.9 (Product category)**.** *Given categories* $\mathcal{C}$ *and* $\mathcal{D}$*, the category* $\mathcal{C} \times \mathcal{D}$ *is defined by the following data:*

- $|\mathcal{C} \times \mathcal{D}| :\equiv |\mathcal{C}| \times |\mathcal{D}|$ *where* $\times$ *is the product defined for types*
- $Hom_{\mathcal{C} \times \mathcal{D}}((C, D), (C', D')) :\equiv Hom_{\mathcal{C}}(C, C') \times Hom_{\mathcal{D}}(D, D')$
- $(f', g') \circ (f, g) :\equiv (f' \circ f, g', \circ g)$
- $1_{(C,D)} :\equiv (1_C, 1_D)$

It is easy to show that the laws hold for the above data.

2.2.2 Functors

A functor is a structure-preserving mapping between categories, both for objects and morphisms. Both operations are denoted by juxtaposition.

Definition 2.10 (Functor)**.** *A functor $F : \mathcal{C} \to \mathcal{D}$ is given by the following data:*

- *for all $A : |\mathcal{C}|$ an object $FA : |\mathcal{D}|$*
- *for all $f : Hom_{\mathcal{C}}(A, B)$ a morphism $Ff : Hom_{\mathcal{D}}(FA, FB)$*

obeying the following laws:

- *for all f, g composable, $F(f \circ g) = Ff \circ Fg$*
- *for all $A : |\mathcal{C}|$, $F\, 1_A = 1_{FA}$*

A functor $F : \mathcal{C}^{op} \to \mathcal{D}$ is called a *contravariant functor* from $\mathcal{C}$ to D. The difference from a normal (covariant) $\mathcal{C} \to \mathcal{D}$ functor is that if $f : A \to B$, then $Ff : FB \to FA$, and $F(f \circ g) = Fg \circ Ff$.

A functor $F : \mathcal{C} \to \mathcal{C}$ is called an *endofunctor*.

A functor $F : \mathcal{C}^{op} \to \mathbf{Set}$ is called a *presheaf*.

Definition 2.11 (Hom functor)**.** *Given a category $\mathcal{C}$, we can define the Hom-sets as a functor $Hom_{\mathcal{C}} : \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$ by the following data:*

- *$Hom_{\mathcal{C}}$ applied to an object (A, B) is defined as the Hom-set $Hom_{\mathcal{C}}(A, B)$*
- *if $f : A \to B$, $g : C \to D$ then $Hom_{\mathcal{C}}(f, g) : Hom_{\mathcal{C}}(B, C) \to Hom_{\mathcal{C}}(A, D)$, $Hom_{\mathcal{C}}(h) :\equiv g \circ h \circ f$*

Definition 2.12 (Diagonal functor)**.** *For any category $\mathcal{C}$, the functor $\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ is defined by:*

- *$\Delta A :\equiv (A, A)$*
- *if $f : A \to B$, $\Delta f :\equiv (f, f)$*

Definition 2.13 (Constant functor)**.** *The constant functor $const_D : \mathcal{C} \to \mathcal{D}$ maps every object in $\mathcal{C}$ to $D : |\mathcal{D}|$ and every morphism to 1_D.*

2.2.3 Natural transformations

A natural transformation is a mapping between functors having the same type. They map objects in the domain category of the functors into arrows arrows in the target category.

Definition 2.14 (Natural transformation)**.** *If $F, G : \mathcal{C} \to \mathcal{D}$ functors, a natural transformation $\alpha : F \dot{\to} G$ is a collection of arrows $\alpha_A : FA \to GA$ for all $A : |\mathcal{C}|$ s.t. for all $f : Hom_{\mathcal{C}}(A, B)$ the following diagram commutes:*

$$\begin{tikzcd}
FA \arrow[r, "\alpha_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\alpha_B"] & GB
\end{tikzcd}$$

That is, $Gf \circ \alpha_A = \alpha_B \circ Ff$.

Definition 2.15 (Natural isomorphism). *Given $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\alpha : F \to G$ is a natural isomorphism if for all $A : |\mathcal{C}|$, $\alpha_A : FA \to GA$ is an isomorphism.*

Definition 2.16 (Functor category). *Given categories $\mathcal{C}$, $\mathcal{D}$, the functor category $\mathcal{D}^{\mathcal{C}}$ is defined by:*

- *objects are functors of type $\mathcal{C} \to \mathcal{D}$*
- *morphisms are natural transformations*
- *composition is the pointwise: if $\alpha : F \dot{\to} G$, $\beta : G \dot{\to} H$, then $(\beta \circ \alpha)_A :\equiv \beta_A \circ \alpha_A$*
- *identity is the identity natural transformation: $1_F : F \dot{\to} F$, $(1_F)_A :\equiv 1_{FA}$*

2.2.4 Adjunctions

Definition 2.17 (Adjunction). *Given functors : $F : \mathcal{C} \to \mathcal{D}$, $G : \mathcal{D} \to \mathcal{C}$, we say that F is left adjoint to G (or G is right adjoint to F), written as $F \dashv G$ if there is an isomorphism*

$$Hom_{\mathcal{D}}(FC, D) \cong Hom_{\mathcal{C}}(C, GD)$$

natural in all $C : |\mathcal{C}|$, $D : |\mathcal{D}|$.

*This means that for the functors $Hom_{\mathcal{D}}(F\cdot, D), Hom_{\mathcal{C}}(\cdot, GD) : \mathcal{C} \to$ **Set** there should be a natural isomorphism $\alpha : Hom_{\mathcal{D}}(F\cdot, D) \dot{\to} Hom_{\mathcal{C}}(\cdot, GD)$. Similarly there should be a natural isomorphism $\beta : Hom_{\mathcal{D}}(FC, \cdot) \dot{\to} Hom_{\mathcal{C}}(C, G\cdot)$.*

Definition 2.18 (Product and coproduct). *A category $\mathcal{C}$ has products if Δ has a right adjoint called the product functor $\cdot \times \cdot : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. Dually, $\mathcal{C}$ has coproducts if Δ has a left adjoint, the coproduct functor $\cdot + \cdot : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. In summary: $\cdot + \cdot \dashv \Delta \dashv \cdot \times \cdot$.*

This means the following natural isomorphism for the product functor:

$$Hom_{\mathcal{C}}(C, A) \times Hom_{\mathcal{C}}(C, B) \equiv Hom_{\mathcal{C}\times\mathcal{C}}(\Delta C, (A, B)) \overset{\overset{<\cdot,\cdot>}{\longrightarrow}}{\underset{\underset{<\pi_1\circ\cdot,\pi_2\circ\cdot>}{\longleftarrow}}{\cong}} Hom_{\mathcal{C}}(C, A \times B)$$

π_1, π_2 are the projections, $< \cdot, \cdot >$ is called split.

And the following natural isomorphism for the coproduct functor:

$$Hom_{\mathcal{C}}(A+B, C) \overset{\overset{<\cdot\circ\iota_1,\cdot\circ\iota_2>}{\longrightarrow}}{\underset{\underset{[\cdot,\cdot]}{\longleftarrow}}{\cong}} Hom_{\mathcal{C}\times\mathcal{C}}((A, B), \Delta C) \equiv Hom_{\mathcal{C}}(A, C) \times Hom_{\mathcal{C}}(B, C)$$

ι_1, ι_2 are the injections, $[\cdot, \cdot]$ is called join.

Definition 2.19 (Initial and terminal object). *A category $\mathcal{C}$ has a terminal object if $const_* : \mathcal{C} \to \mathbf{1}$ has a right adjoint $term : \mathbf{1} \to \mathcal{C}$ and the terminal object is $term *$. Dually, $\mathcal{C}$ has an initial object $init *$ if $const_*$ has a left adjoint.*

These mean the following isomorphisms:

$$Hom_{\mathbf{1}}(*, *) \equiv Hom_{\mathbf{1}}(const_* A, *) \cong Hom_{\mathcal{C}}(A, term\, *)$$

$$Hom_{\mathcal{C}}(init\, *, A) \cong Hom_{\mathbf{1}}(*, const_*\, A) \equiv Hom_{\mathbf{1}}(*, *)$$

That is, if $\mathcal{C}$ has a terminal object, we have an arrow $A \to term\, *$ for all $A : |\mathcal{C}|$. Dually, if there is an initial object, we have an arrow $init\, * \to A$ for all $A : |\mathcal{C}|$.

Definition 2.20 (Exponential)**.** *A category $\mathcal{C}$ has exponentials (or internal Hom-sets) if it has products and, for any $A : |\mathcal{C}|$, the partially applied product functor $\cdot \times A : \mathcal{C} \to \mathcal{C}$ has a right adjoint $A \Rightarrow \cdot : \mathcal{C} \to \mathcal{C}$. This means:*

$$Hom_{\mathcal{C}}(C \times A, B) \underset{\overleftarrow{uncurry}}{\overset{\overrightarrow{curry}}{\cong}} Hom_{\mathcal{C}}(C, A \Rightarrow B)$$

We define a morphism $eval_{AB} : (A \Rightarrow B) \times A \to B$ by $eval_{AB} :\equiv uncurry(1_{A \Rightarrow B})$.

If $\mathcal{C}$ has exponentials, we can extend the exponential functor to $\cdot \Rightarrow \cdot : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ which acts on objects as described above, and given $f : A \to B$, $g : C \to D$, $f \Rightarrow g : (B \Rightarrow C) \to (A \Rightarrow D)$, $f \Rightarrow g :\equiv curry(g \circ eval_{BC} \circ (1_{B \Rightarrow C} \times f))$.

Definition 2.21 (Cartesian closed category)**.** *A category is called cartesian closed if it has a terminal object, products and exponentials.*

2.2.5 Higher categories

We can define a category of categories **Cat**. This has categories as objects (we need smallness, so **Cat** can't be an object of itself), and functors as morphisms. But we have a mapping between functors as well which is not described by the structure of a category. This leads us to the notion of a higher category, where we have morphisms between morphisms as well. At the same time, while having higher morphisms, we can weaken the identity and associativity rule to say that $id \circ f$ is not equal, but isomorphic to f.

In normal category theory, we can speak about equality of objects by two means:

1. stating that two objects are simply equal (using the equality of our metatheory)
2. stating that two objects are isomorphic (using the built-in notion of objects)

The first notion is called "evil" and definitions using such an evil notions are disallowed because they don't respect the second equality, only the first. We only have one way to talk about equality of morphisms, the meta-theoretic equality. However, if we have higher morphisms between morphisms, we have the second type of equality, and the first type will be considered evil.

Definition 2.22 (2-category)**.** *A (weak) 2-category is given by the following data:*

- *a type of objects which are called 0-cells*

- *for any two objects A, B, a category $Hom(A, B)$ the objects of which are called 1-cells, the morphisms of which are called 2-cells*
- *for each 0-cell A, a 1-cell $1_A : |Hom(A, A)|$*
- *for each triple of 0-cells A, B, C, a functor $\cdot\circ\cdot : Hom(B, C) \times Hom(A, B) \to Hom(A, C)$*
- *for each pair of 0-cells A, B the following natural isomorphisms*[7] *called unitors:*

$$\begin{aligned}1_{Hom(A,B)} \circ const_{1_A} &\cong 1_{Hom(A,B)} \\ &\cong const_{1_B} \circ 1_{Hom(A,B)} : Hom(A, B) \to Hom(A, B)\end{aligned}$$

- *for each four 0-cells A, B, C, D, a natural isomorphism called associator:*

$$\begin{aligned}(h, g, f) \mapsto h \circ (g \circ f) &\cong (h, g, f) \mapsto (h \circ g) \circ f \\ &: Hom(C, D) \times Hom(B, C) \times Hom(A, B) \to Hom(A, D)\end{aligned}$$

  such that using the associator clockwise or counter-clockwise in the following pentagon to create a natural isomorphism from the left-associating to the right associating functors result in pointwise equal natural transformations.

$$\begin{tikzcd}
 & {(f,g,h,i) \mapsto (f \circ g) \circ (h \circ i)} \arrow[dr, "\cong"] & \\
{(f,g,h,i) \mapsto ((f \circ g) \circ h) \circ i} \arrow[ur, "\cong"] \arrow[d, "\cong"] & & {(f,g,h,i) \mapsto f \circ (g \circ (h \circ i))} \\
{(f,g,h,i) \mapsto (f \circ (g \circ h)) \circ i} \arrow[rr, "\cong"] & & {(f,g,h,i) \mapsto f \circ ((g \circ h) \circ i)} \arrow[u, "\cong"']
\end{tikzcd}$$

From the unitors it follows that given an $f : Hom(A, B)$, $f \circ 1_A \cong f$, since

$$\begin{aligned}& f \circ 1_A \\ =& 1_{Hom(A,B)}(f) \circ const_{1_A}(f) \\ =& (1_{Hom(A,B)} \circ const_{1_A})(f) \\ \cong& 1_{Hom(A,B)}(f) \\ =& f\end{aligned}$$

Similarly, we get the other identity law and associativity up to isomorphism.

So, in addition to the identity and associativity laws for categories we have a higher coherence law for associativity itself. The situation becomes much more complicated when generalizing further, to n-categories, where the laws for

[7] $1_{Hom(A,B)}$ is the identity functor $f \mapsto f$, $const_{1_A}$ is the constant functor $h \mapsto 1_A$

k-cells are given up to $(k+1)$-cells if $k < n$. The most generalized notion is ∞-category (or ω-category[8]) where there is no upper limit of levels.

2.2.6 Applications

Category theory provides a high level language to talk about type theories. We give some examples of its usage in this area:

- Cartesian closed categories and the simply typed lambda calculus are the same.
- A categorical model of type theory is given in section 3.
- Types together with their identity types can be viewed as (weak) ∞-groupoids (see section 2.3).
- The theory of data types (see section 2.1.10) can be given in category theory: a container is a polynomial functor and a W-type is an initial objects in the category of algebras of that functor. Dually, a coinductive type is a terminal object in the category of coalgebras for that functor. (An algebra for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is an object $A : |\mathcal{C}|$ and a morphism $FA \to A$. Objects in the category of algebras are algebras while a morphism from $f : FA \to A$ to $g : FB \to B$ is a morphism $h : A \to B$ s.t. $h \circ f = g \circ Fh$.) The approach can be extended to
- Monads are endofunctors with additional properties. They provide a way to describe computations with side effects [64] and to define the order of evaluation in functional programming.

2.2.7 Notes

Category theory was founded by Eilenberg and MacLane in 1945 [25]. An introductory book for computer scientist is [56], a more detailed introduction is [7] and the most standard textbook is [41].

An introduction to the connection between category theory and logic is [38].

The Homotopy Type Theory book [58] has a chapter on formalizing category theory in type theory, giving a definition of category which does not allow "evil" statements.

2.3 Homotopy Type Theory

2.3.1 Homotopies

Traditionally types in type theory are viewed as sets. The propositions as types view [35] does not change this as it describes a type (proposition) as the set of its proofs.

However, the equality type does not really behave as expected for sets: the naturally looking rule saying that every proof of equality is equal to refl (since this is its constructor) is not provable by the usual elimination rule J of equality viewed as an inductive type (the K rule, see 2.1.10). This was proved by Hofmann and Streicher by constructing the groupoid model where K is false [34].

[8]We only consider weak versions, so this is to be understood as weak ∞-category as opposed to strict ∞-category.

A generalisation of this result was given by Awodey [8] and Voevodsky [62] who interpret types as topological spaces, or, by the homotopy hypothesis, ∞-groupoids. Viewing a type as a space, an element of the type is a point in the space, an equality between two elements is a path in the space, an equality between two equalities of two elements is a homotopy between the two paths. We make this more precise:

Definition 2.23 (Homotopy (homotopy theory))**.** *If $X, Y \subseteq \mathbb{R}$, a homotopy between the continuous functions $f, g : X \to Y$ is a continuous function $h : [0,1] \times X \to Y$ such that for all $x : X \,.\, h(0,x) = f(x)$ and $h(1,x) = g(x)$. Notation: $f \sim g$.*

A homotopy can be thought of as the continuous deformation of the image of f into the image of g. With this definition, we have the following correspondence between homotopies in homotopy theory and equalities in type theory. The first three levels of homotopies in homotopy theory are as follows:

$$\begin{aligned}
&f : [0,1] \to X \text{ continuous},\ f(0) = a,\ f(1) = b\\
&h : [0,1]^2 \to X \text{ cont.},\ \forall z : [0,1] \,.\, h(0,z) = f(z),\ h(1,z) = g(z)\\
&j : [0,1]^3 \to X \text{ cont.},\ \forall z : [0,1]^2 \,.\, j(0,z) = h(z),\ j(1,z) = i(z)
\end{aligned}$$

The corresponding equalities in type theory:

$$\begin{aligned}
&f : \sum_{x,x':X} x =_X x',\ \mathsf{pr}_1(\mathsf{pr}_1(f)) = a, \mathsf{pr}_2(\mathsf{pr}_1(f)) = b\\
&h : \sum_{p,q:\left(\sum_{x,x':X} x =_X x'\right)} p = q,\ \mathsf{pr}_1(\mathsf{pr}_1(h)) = f,\ \mathsf{pr}_2(\mathsf{pr}_1(h)) = g\\
&j : \sum_{r,s:\left(\sum_{p,q:\left(\sum_{x,x':X} x =_X x'\right)} p = q\right)} r = s,\ \mathsf{pr}_1(\mathsf{pr}_1(j)) = h,\ \mathsf{pr}_2(\mathsf{pr}_1(j)) = i
\end{aligned}$$

Reflexivity of equality is given by the constant path, symmetry by precomposing a homotopy by the $[0,1] \to [0,1]$ function $z \mapsto 1 - z$, transitivity given an f, g where $f(1) = g(0)$ by

$$(f \bullet g)(z) :\equiv \begin{cases} f(z * 2) & z < \frac{1}{2}\\ g\left(\left(z - \frac{1}{2}\right) * 2\right) & \text{otherwise}\end{cases}$$

Two functions being pointwise equal now can be thought of as a homotopy between the two functions (a deformation of one image of the function to the other image by paths, which are equalities in type theory):

Definition 2.24 (Homotopy (type theory))**.** *A homotopy between two functions $f, g : \Pi\, A\, B$ is defined by:*

$$f \sim g :\equiv \Pi\, A \left(\mathsf{app}(f, \mathsf{q}) =_{B[x]} \mathsf{app}(g, \mathsf{q})\right)$$

More generically:

$$\cdot \hat{\sim} \cdot : \Pi_{\,\mathsf{U}} \left(\Pi_{\,(\mathsf{El}(\mathsf{q})\to\mathsf{U})} \left(\mathsf{El}(\hat{\Pi}\,\mathsf{qp}\,\mathsf{app}(\mathsf{qp},\mathsf{q})) \to \mathsf{El}(\hat{\Pi}\,\mathsf{qp}\,\mathsf{app}(\mathsf{qp},\mathsf{q})) \to \mathsf{U}\right)\right)$$

$$\cdot \hat{\sim} \cdot :\equiv {}_{\lambda\,\lambda}\,\lambda\lambda\hat{\Pi}\,\mathsf{qppp}\,(\mathsf{app}(\mathsf{qpp},\mathsf{q})\hat{=}_{\mathsf{qppp}[\mathsf{q}]}\mathsf{app}(\mathsf{qp},\mathsf{q}))$$

Definition 2.25 (Relative homotopy)**.** *If $f, g : X \to Y$, $A \subseteq X$, a homotopy relative to A is a $h : f \sim g$ s.t. for all $a : A\,.\,h(t,a)$ is independent of t.*

This definition only makes sense if $f\big|_A = g\big|_A$. A relative homotopy is a continuous deformation of the images of the functions where the parts which are elements of A are constant i.e. do not move. A homotopy relative to $\{0,1\} \subseteq [0,1]$ between $f, g : [0,1] \to X$ where $f(0) = a = g(0)$ and $f(1) = b = g(1)$ can be also written as:

$$h : [0,1] \to \sum_{p:[0,1]\to X} p(0) = a \times p(1) = b \qquad \text{where } h(0) = f \text{ and } h(1) = g$$

In type theory, we would write

$$a : X, b : X, f : a = b, g : a = b \vdash h : f = g$$

The categorical take on the types as spaces view says that types are ∞-groupoids. An element of a type is an object (0-cell) of the ∞-groupoid, an equality between two elements is a morphism (1-cell) between the two objects (which is an isomorphism, as all morphisms in a groupoid are isomorphisms), an equality between two equalities is a 2-cell etc.

Just as types can be seen as groupoids, functions are functors between ∞-groupoids. The object part of the functor is simple function application, while the morphism part takes a morphisms in X of type $x =_X x'$ for some $x, x' : X$ to a morphism of type $\mathsf{app}(f,x) =_Y \mathsf{app}(f,y)$. This is usually called ap (action on paths):

$$\begin{aligned}\mathsf{ap}(\cdot,\cdot) : \Pi_{\,\mathsf{U}} \Bigg(\Pi_{\,\mathsf{U}} \bigg(\Pi \left(\mathsf{El}(\mathsf{qp}) \to \mathsf{El}(\mathsf{q})\right) \\ \left(\Pi_{\,\mathsf{El}(\mathsf{qpp})} \left(\Pi_{\,\mathsf{El}(\mathsf{qp}^3)} (\mathsf{qp} = \mathsf{q} \to \mathsf{app}(\mathsf{qpp},\mathsf{qp}) = \mathsf{app}(\mathsf{qpp},\mathsf{q}))\right)\right)\bigg)\Bigg)\end{aligned}$$

$$\mathsf{ap}(\cdot,\cdot) :\equiv {}_{\lambda\,\lambda}\,\lambda\,{}_{\lambda\,\lambda}\,\lambda\mathsf{J}(\mathsf{refl}_{\mathsf{app}(\mathsf{qp}^4,\mathsf{q})}, \mathsf{qpp}, \mathsf{qp}, \mathsf{q})$$

2.3.2 Homotopy levels

Definition 2.1 of sets (types where all equalities between two elements are equal) corresponds to that of discrete spaces in homotopy theory or 0-groupoids in category theory. Similarly, types for which equalities of equalities are equal are called homotopy 1-types in homotopy theory or 1-groupoids (or just groupoids) in category theory. We can generalize this as follows:

Definition 2.26 (Contractible)**.** *A type X is contractible if*

$$\mathsf{isContr}(X) :\equiv \sum_{x:X} \left(\prod_{x':X} x = x'\right)$$

is inhabited. More precisely: the type having code $\hat{X}$ *is contractible if* $\mathsf{El}(\mathsf{is}\widehat{\mathsf{Contr}}(\hat{X}))$ *is inhabited where* $\mathsf{is}\widehat{\mathsf{Contr}}$ *is defined as follows:*

$$\begin{aligned}&\mathsf{is}\widehat{\mathsf{Contr}}(\cdot) : \mathsf{U} \to \mathsf{U}\\ &\mathsf{is}\widehat{\mathsf{Contr}}(\cdot) :\equiv \lambda\hat{\Sigma}\,\mathsf{q}\,\big(\hat{\Pi}\,\mathsf{qp}\,(\mathsf{qp}\hat{=}_{\mathsf{qpp}}\mathsf{q})\big)\end{aligned}$$

Definition 2.27 (h-level)**.** *If* $n \geq -2$*, we define homotopy levels as follows:*

$$\begin{aligned}&\mathsf{is}\text{-}\widehat{n\text{-}}\mathsf{type}(\cdot) : \mathsf{U} \to \mathsf{U}\\ &\mathsf{is}\text{-}(-\hat{2})\text{-}\mathsf{type}(\cdot) :\equiv \mathsf{is}\widehat{\mathsf{Contr}}(\cdot)\\ &\mathsf{is}\text{-}(n\,\hat{+}\,1)\text{-}\mathsf{type}(\cdot) :\equiv \lambda\hat{\Pi}\,\mathsf{q}\,\big(\hat{\Pi}\,\mathsf{qp}\,(\mathsf{is}\text{-}\widehat{n\text{-}}\mathsf{type}(\mathsf{qp}\hat{=}_{\mathsf{qpp}}\mathsf{q}))\big)\end{aligned}$$

h-level -1 is also called the level of (mere) propositions — types which do not carry any information apart from being inhabited or not. For example, the type $\mathsf{is}\text{-}n\text{-}\mathsf{type}(X)$ is a proposition for any n. h-level 0 corresponds to sets (we denote the type of sets by **Set**), h-level 1 to groupoids etc. This gives an internal notion of propositions in the type theory instead of an externally given proof-irrelevant universe such as Prop in Coq [46].

h-levels are cumulative which means that if a type is of h-level n, it is also of h-level m for all $m > n$.

h-levels are not to be confused with universe levels. However, they are not completely independent: by showing that all type formers preserve h-levels and base types (0, 1, 2) have h-level 0, we know that all constructible types in U_0 are of h-level 0 (this is a meta-theoretic statement not provable within the theory). This can be generalized to higher levels, that is, all constructible types in U_n are n-types (metatheoretically). It was shown (within the theory) by Kraus and Sattler that the universe U_n itself is not an n-type [37]. We summarize this in the following table giving examples of types where they can be constructed and writing "–" where there are no constructible types.

		U_0	U_1	U_2	U_3	...
	0	$\hat{2}$	$\hat{\mathsf{U}}_0\hat{\to}\hat{2}$	$\hat{\mathsf{U}}_1\hat{\to}\hat{2}$	$\hat{\mathsf{U}}_2\hat{\to}\hat{2}$	
	1	–	$\hat{\mathsf{U}}_0$	$\hat{\mathsf{U}}_1\hat{\to}\hat{\mathsf{U}}_0$	$\hat{\mathsf{U}}_2\hat{\to}\hat{\mathsf{U}}_0$	
h-level	2	–	–	$\hat{\mathsf{U}}_1$	$\hat{\mathsf{U}}_2\hat{\to}\hat{\mathsf{U}}_1$	
	3	–	–	–	$\hat{\mathsf{U}}_2$	
	...					...

We used cumulativity of universes mentioned in section 2.1.8 in the case of codes like $\hat{\mathsf{U}}_1\hat{\to}\hat{\mathsf{U}}_0$ (we need $\hat{\mathsf{U}}_0 : \mathsf{U}_2$ to use the Π of U_2 to construct the code $\hat{\mathsf{U}}_1\hat{\to}\hat{\mathsf{U}}_0 : \mathsf{U}_2$).

In the presence of higher inductive types (2.3.4) one can construct types in any of the cells in the above table.

2.3.3 Equivalence

Equivalence of spaces in homotopy theory is stated as follows.

Definition 2.28 (Homotopy equivalence)**.** *Given two spaces* X*,* Y*, the continuous function* $f : X \to Y$ *is a homotopy equivalence if there exists a* $g : Y \to X$ *continuous function such that* $f \circ g \sim id_Y$ *and* $g \circ f \sim id_X$*.*

Voevodsky's univalence axiom states that equality of types can be given by equivalences. Equivalences are the type theoretic counterpart of homotopy equivalences with an additional coherence condition making $\mathsf{isEquiv}(f)$ a proposition for all fs.

Definition 2.29 (Equivalence)**.** *Given $\hat{A}, \hat{B} : \mathsf{U}$, an $f : \mathsf{El}(\hat{A} \hat{\to} \hat{B})$, the type stating that f is an equivalence is*

$$\mathsf{is\hat{E}quiv}(f) :\equiv \hat{\sum_{(g:\mathsf{El}(\hat{B}\hat{\to}\hat{A}))}} \hat{\sum_{(\eta:\mathsf{El}(g\circ f\sim id_{\hat{A}}))}} \hat{\sum_{(\epsilon:\mathsf{El}(f\circ g\sim id_{\hat{B}}))}}$$

$$\hat{\prod_{(x:\mathsf{El}(\hat{A}))}} \mathsf{ap}(f, \mathsf{app}(\eta, x)) \mathrel{\hat{=}}_{\mathsf{app}(f,\mathsf{app}(g,\mathsf{app}(f,x))) \mathrel{\hat{=}}_{\hat{B}} \mathsf{app}(f,x)} \mathsf{app}(\epsilon, \mathsf{app}(f, x))$$

We use the following notation:

$$\hat{A} \mathrel{\hat{\simeq}} \hat{B} :\equiv \hat{\Sigma}\, (\hat{A} \mathrel{\hat{\to}} \hat{B})\, \mathsf{is\hat{E}quiv}(\mathsf{q})$$

Isomorphism (called quasi-inverse in [58]) of two types can be simply defined by omitting the last coherence condition — and from an isomorphism one can always derive an equivalence. However, being an isomorphism is not a proposition (h-level -1).

Definition 2.30 (Univalence)**.** *A universe* U *is univalent if the canonical mapping from identity to equivalence is an equivalence:*

$$\mathsf{ua} : \Pi_{\;\hat{A}:\mathsf{U}} \left(\Pi_{\;\hat{B}:\mathsf{U}}\, \mathsf{El}((\hat{A} \mathrel{\hat{=}}_{\hat{\mathsf{U}}} \hat{B}) \mathrel{\hat{\simeq}} (\hat{A} \mathrel{\hat{\simeq}} \hat{B}))\right)$$

The mapping from $p : (\hat{A} \mathrel{\hat{=}}_{\hat{\mathsf{U}}} \hat{B})$ to $(\hat{A} \mathrel{\hat{\simeq}} \hat{B})$ is defined by the function $\mathsf{transport}^{\lambda \mathsf{U}}(p, \cdot)$*, the fact that this is an equivalence can be proven easily by* J.

Univalence justifies both informal mathematical practices mentioned in section 2.1.13: function extensionality and isomorphic types being equal [58], however it lacks a computational interpretation just as those axioms. We will look at possible ways of finding a solution for this problem in section 3.

2.3.4 Higher inductive types

Inductive types (section 2.1.10) are given by constructors which give elements of that type (points, if viewed as a space). The natural generalisation is to specify types where constructors can also give equalities of that type (paths and higher paths in the space). An example is the type $\mathbb{S}^1$ given by the following constructors:

$$\begin{aligned} &\mathsf{base} : \mathbb{S}^1 \\ &\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base} \end{aligned}$$

The notion of eliminator for an inductive type can be generalized to higher inductive types; however the theory of containers is not yet extended to higher inductive types, and there is also no general scheme for deriving the eliminator from the constructors. However, for the above type, the eliminator should be the following:

$$\frac{\Gamma.\mathbb{S}^1 \vdash P \quad \Gamma \vdash b : P[\mathsf{base}] \quad \Gamma \vdash l : \mathsf{transport}^P(\mathsf{loop}, b) =_{P[base]} b \quad \Gamma \vdash x : \mathbb{S}^1}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(b, l, x) : P[x]}$$

That is, we need to give a method b of the family P at base and also a method l which is an equality of b and b lying over loop in P.

The computation rule is $\mathsf{ind}_{\mathbb{S}^1}(b, l, \mathsf{base}) \equiv b$ and $\mathsf{apd}(\lambda \mathsf{ind}_{\mathbb{S}^1}(b, l, \mathsf{q}), \mathsf{loop}) = l$. The second rule is a propositional computation rule using a version of the ap function defined in section 2.3.1 which works for dependently typed functions. The propositional computation rules provide new inhabitants of the identity type without specifying how to eliminate them, so they pose the same problem as univalence.

Higher inductive types can be used to *truncate* other types to some h-level. E.g. propositional truncation creates a mere proposition out of a type from which no other information can be obtained but inhabitedness.

Definition 2.31 (Propositional truncation). *Given a type A, the type $||A||$ is defined as a higher inductive type by the following constructors:*

$$\begin{aligned} |\cdot| &: A \to ||A|| \\ \mathsf{prop\text{-}eq} &: \Pi\,(x, y : ||A||)\,(x = y) \end{aligned}$$

We can only eliminate over this type if we can't distinguish between elements of the motive of the elimination:

$$\frac{\begin{array}{l} \Gamma.||A|| \vdash P \\ \Gamma \vdash g : \Pi\, A\, P[|\mathsf{q}|] \\ \Gamma.||A||.||A\mathsf{p}||.P(\mathsf{pp}, \mathsf{qp}).P(\mathsf{ppp}, \mathsf{qp}) \vdash q : \mathsf{transport}^P(\mathsf{prop\text{-}eq}(\mathsf{qp}^3, \mathsf{qp}^2), \mathsf{qp}) = \mathsf{q} \\ \Gamma \vdash x : ||A|| \end{array}}{\Gamma \vdash \mathsf{ind}_{||A||}(g, q, x) : P[x]}$$

Another useful consequence of higher inductive types is that they give quotient types. For example, given the natural numbers $\mathbb{N}$, integers can be defined by the higher inductive type given by the following constructors:

$$\begin{aligned} &\mathsf{minus}(\cdot, \cdot) : \mathbb{N} \to \mathbb{N} \to \mathbb{Z} \\ &\mathsf{quot} : \prod_{a,b,c,d:\mathbb{N}} a + d = c + b \to \mathsf{minus}(a, b) =_{\mathbb{Z}} \mathsf{minus}(c, d) \\ &\mathsf{set} : \prod_{(x,y:\mathbb{Z})} \prod_{(p,q:x=_{\mathbb{Z}}y)} p =_{x=_{\mathbb{Z}}y} q \end{aligned}$$

The last constructor ensures that we have no higher equalities, i.e. $\mathbb{Z}$ is a set.

2.3.5 Notes

The question whether uniqueness of identity proofs is unprovable from J was raised by [3] in 1992. 4 year later, Hofmann and Streicher constructed a model that refutes it [34]. They suggested the generalisation of their groupoid model to higher groupoids and also noted that "such version of identity sets may be useful for a formulation of category theory inside type theory providing a formal underpinning for the common practice of considering isomorphic objects as equal." The connection with topology was discovered independently by Voevodsky [62] and Awodey and Warren [8] in 2006. The univalence axiom was introduced in [36]. These discoveries culminated in a special year on Univalent Foundations Institute of Advanced Study in Princeton 2012/2013, where the book *Homotopy Type Theory: Univalent Foundations of Mathematics* [58], an informal introduction for mathematicians was written.

3 Models of Type Theory

Having an algorithm which computes the normal form from any term is a requirement for the type theory to serve as a programming language. One could assert univalence and the propositional computation rules for higher inductive types as axioms, but then any computation involving these would have to be done manually. Another worry could be that these additional axioms might make the theory inconsistent - however, this is not the case as showed by a model construction by Voevodsky [36].

One could consider the following two ways to construct an algorithm for normalisation:

- Trying to figure out the elimination rules for the additional elements of equality and showing normalisation by a technique mentioned in section 2.1.12.
- Using model construction. A model in general is a sound semantics, that is, a specification of the true judgments such that for all rules of the formal system, if the premises[9] are true, then so is the conclusion. In the next section, we will give a definition of model specific to type theory which interprets each context, type and term in a semantic domain with the special property that definitionally equal terms are interpreted as the same semantic objects. If there is a mapping from the semantic term objects back to the syntactic terms and the model construction is done constructively, we are able to derive a normalisation algorithm from it: we need to compose the interpretation function by the function mapping semantic values back to syntactic values and define such syntactic values as normal forms. This method is called normalisation by evaluation [11].

We try to follow the second method. The proposed model should satisfy all the rules of type theory together with the additional rules of homotopy type theory: univalence and propositional computation rules for higher inductive types. One way to implement such a model is to use type theory as a metatheory and use the computational properties of this metatheory to derive the computation properties of the object theory. To achieve this, syntactic objects (terms) which are definitionally equal in the object theory should be represented by semantic objects which are definitionally equal in the metatheory. If this is the case, definitionally equal terms would be normalised automatically to the same terms. In what follows, we present our definition of model, give a list of some models described in the literature, then turn our attention to the Kan semisimplicial set model by Thierry Coquand [10].

3.1 Categories with families

The definition of model of type theory that we will use is Peter Dybjer's categories with families [22]. To distinguish semantic constructs from the metatheory's syntax we use informal language or the high level notations as introduced in section 2.1 when using metatheoretic constructs. For example, instead of saying that we have a dependent function $f : \Pi\ A\ B$, we write: for each $a : A$

[9]Because the premises usually contain variables we need to take all possible instances of each rule into account.

we have an $f(a) : B(a)$. Function application in the metatheory is expressed by $f(a)$ instead of $\mathsf{app}(f, a)$. Dependent sums are expressed as $(x : A) \times B(x)$. $=$ refers to metatheoretic equality (which might be propositional or definitional in the metatheory). A more principled way would have been to use a logical framework [31] notation to distinguish the object and the metatheory.

Definition 3.1 (Category of families). *The category of families Fam is given by the following data:*

- *objects: pairs (A, A') where A is a set, A' is a family of sets indexed over A, i.e. if $a : A$, then $A'(a)$ is a set*
- *a morphism from (A, A') to (B, B') is a pair of functions (f, f') s.t. if $a : A$ and $a' : A'(a)$ then $f(a) : B$ and $f'(a') : B'(f(a))$*
- *composition and identities are defined in the straightforward way*

Definition 3.2 (Categories with families (CwF)). *A category with families is given by the following data:*

- *A category $\mathcal{C}$ with a terminal object (contexts and context morphisms).*
- *A functor $F : \mathcal{C}^{op} \to Fam$. Notation: if $\Gamma : |\mathcal{C}|$ we write $F\Gamma$ as $(Ty(\Gamma), Tm(\Gamma, \cdot))$. If $\sigma : \Delta \to \Gamma$ then $(\cdot\, \sigma, \cdot\, \sigma) :\equiv F\sigma$ and if $A : Ty(\Gamma)$ and $t : Tm(\Gamma, A)$ then $A\sigma : Ty(\Delta)$, $t\sigma : Tm(\Delta, A\sigma)$.*
- *For each $\Gamma : |\mathcal{C}|$ and $A : Ty(\Gamma)$ a $\Gamma.A : |\mathcal{C}|$ called the comprehension of Γ and A, and the functions $(\cdot, \cdot)$, $\mathsf{p}\,\cdot$ and $\mathsf{q}\,\cdot$ which express that the following is an isomorphism for all Δ, Γ, A:*

$$\big(\sigma : Hom_{\mathcal{C}}(\Delta, \Gamma)\big) \times \big(Tm(\Delta, A\sigma)\big) \underset{\xleftarrow{<\mathsf{p}\,\cdot, \mathsf{q}\,\cdot>}}{\overset{\xrightarrow{(\cdot,\cdot)}}{\cong}} Hom_{\mathcal{C}}(\Delta, \Gamma.A)$$

This definition expresses the rules of type theory given in section 2.1.3 and 2.1.4 (we don't need semantic counterparts of the congruence rules since definitional equality is modelled by the equality of the metatheory which should have all the congruence properties by default).

Definition 3.3 (Dependent products as in section 2.1.5). *A CwF has dependent products if for all $\Gamma : |\mathcal{C}|, A : Ty(\Gamma)$ and $B : Ty(\Gamma.A)$ there is a $\Pi\, A\, B : Ty(\Gamma)$ for which the following equation holds:*

$$(\Pi\, A\, B)\sigma = \Pi\, A\sigma\, B(\sigma\mathsf{p}, \mathsf{q})$$

And for all Γ, A, B we have the following isomorphism which is natural in Γ:

$$Tm(\Gamma.A, B) \underset{\xleftarrow{\mathsf{app}'}}{\overset{\xrightarrow{\lambda}}{\cong}} Tm(\Gamma, \Pi\, A\, B)$$

If the above is not an isomorphism, there are only the two functions λ and app', but we have the equalities $\mathsf{app}'(f)(1, u)\sigma = \mathsf{app}'(f\sigma)(1, u\sigma)$ and $\mathsf{app}'((\lambda b)\sigma)(1, u) = b(\sigma, u)$, the CwF has a weak Π.

The isomorphism being natural means that λ is a natural transformation from $Tm(\cdot, B) \circ (\cdot.A)$ to $Tm(\cdot, \Pi\, A\, B)$ and similarly for app'. The usual app can be recovered from app' by $\mathsf{app}(f, u) :\equiv \mathsf{app}'(f)(1, u)$.

Definition 3.4 (Dependent sums as in section 2.1.7)**.** *A CwF has dependent sums if for all $\Gamma : |\mathcal{C}|, A : Ty(\Gamma)$ and $B : Ty(\Gamma.A)$ there is a $\Sigma\, A\, B : Ty(\Gamma)$ for which the following equation holds:*

$$(\Sigma\, A\, B)\sigma = \Sigma\, A\sigma\, B(\sigma\mathsf{p}, \mathsf{q})$$

And for all Γ, A, B the following isomorphism holds and is natural in Γ:

$$\big(a : Tm(\Gamma, A)\big) \times Tm(\Gamma, B(1, a)) \quad \underset{\overset{\longleftarrow}{<\mathsf{pr}_1\,\cdot,\mathsf{pr}_2\,\cdot>}}{\overset{\overset{(\cdot,\cdot)}{\longrightarrow}}{\cong}} \quad Tm(\Gamma, \Sigma\, A\, B)$$

Similar definitions can be given for the equality type, universes and inductive types. The definitions correspond exactly to those given in the section describing the rules of type theory 2.1.

3.2 List of models

The groupoid model of type theory [34] interprets types as groupoids and the identity type as the discrete groupoid of all arrows between two objects. So, an interpretation of a type by a groupoid having two different arrows between two objects refutes uniqueness of identity proofs. However, equalities of equalities have no structure in this model, they are either the empty groupoid or the one-element groupoid. The 2-groupoid model [45] has equalities between equalities, but the third level of equalities is trivial. These models were extended to the strict ∞-case by Warren [65] and to the weak ∞-groupoid case by Voevodsky [36]. We give a list of models of type theory containing these and some related ones.

object theory	**model**	**metatheory**
MLTT with a universe closed under Π	a model which does not fit into CwF because definitional equality is interpreted by a separate type former [15]	MLTT
MLTT with function extensionality and UIP	observational type theory (OTT) [5]	MLTT with a proof-irrelevant Prop universe
MLTT (without some definitional rules for projections) with function extensionality and quotient types	setoid model by Hofmann [32]	MLTT
weak MLTT with a univalent universe of propositions	setoid model by Coquand [18]	MLTT
MLTT with a univalent universe of propositions and sets	groupoid model [34]	MLTT
MLTT with a univalent universe of propositions, sets and large sets	2-groupoid model [45]	MLTT (Coq)
MLTT	strict ω-groupoid model [65]	ZFC
MLTT with univalence for all universes	Voevodsky's Kan simplicial set model [36] (∞-groupoid model)	ZFC (uses non-constructive properties, see [19])
MLTT	Semisimplicial set model [10]	constructive parts of ZFC
weak MLTT with a univalent universe of sets	truncated Kan semisimplicial set model [10]	MLTT
weak MLTT with univalence for all universes	Kan semisimplicial set model [10]	constructive set theory

We are interested in studying and implementing Thierry Coquand's Kan semisimplicial set model. First we describe the (simpler) semisimplicial set model.

3.3 Semisimplicial set model

3.3.1 Semisimplicial sets

Definition 3.5 (Semisimplicial set). *Let Δ_+ be the category with objects $[n] = \{0, 1, \ldots, n\}$ non-empty finite linear orders and morphisms injective strictly increasing maps. A semisimplicial set is a presheaf $\Delta_+^{op} \to \mathbf{Set}$.*

Every non-identity injective function is a composition of injective functions leaving out the i^{th} element: $\epsilon^i : [n] \to [n+1]$.

By unfolding the definition of semisimplicial set we get the following structure:

- For the object part of the functor we take sets $X[0]$, $X[1]$ etc. as the images of objects $[0]$, $[1]$ etc.
- For the morphism part of the functor it is enough to give the images of the ϵ^i functions (and identities) and generate the images of compositions of injections by the composition of images. Hence, we take functions $d_i : X[n] \to X[n-1]$ $(i = 0, ..., n)$ as the image of each non-identity injection and the identity function as the image of the identity injection.

We can define the same structure in type theory by saying that $X[0]$ is a type, $X[1]$ is a type family indexed over two elements of $X[0]$ where the images of ϵ^i functions are expressed in the types, $X[2]$ is the type family indexed over elements of $X[1]$ which are in turn indexed over "elements of $X[0]$" etc. The first three levels are listed below:

$$
\begin{aligned}
&X[0] : \mathbf{Set} \\
&X[1] : X[0] \to X[0] \to \mathbf{Set} \\
&X[2] : {\scriptstyle (x_0\, x_1\, x_2 : X[0])} \to X[1]\, x_0\, x_1 \to X[1]\, x_0\, x_2 \to X[1]\, x_1\, x_2 \to \mathbf{Set} \\
&X[3] : {\scriptstyle (x_0\, x_1\, x_2\, x_3 : X[0])} \\
&\qquad\quad {\scriptstyle (x_{01} : X[1]\, x_0\, x_1)(x_{02} : X[1]\, x_0\, x_2)(x_{03} : X[1]\, x_0\, x_3)} \\
&\qquad\quad {\scriptstyle (x_{12} : X[1]\, x_1\, x_2)(x_{13} : X[1]\, x_1\, x_3)(x_{23} : X[1]\, x_2\, x_3)} \\
&\qquad\quad \to X[2]\, x_{01}\, x_{02}\, x_{12} \to X[2]\, x_{01}\, x_{03}\, x_{13} \\
&\qquad\quad \to X[2]\, x_{02}\, x_{03}\, x_{23} \to X[2]\, x_{12}\, x_{13}\, x_{23} \to \mathbf{Set}
\end{aligned}
$$

The full series hasn't been formalised yet in type theory. The geometric interpretation of the series is as follows: $X[0]$ is the set of points (0-dimensional simplices), $X[1]$ is the set of directed edges (1-dimensional simplices) parameterised by two points, for example an edge of type $X[1]\, x_0\, x_1$ has starting point x_0 and end point x_1. An element of type $X[2]\ x_{01}\ x_{02}\ x_{12}$ is a triangle (2-dimensional simplex) with sides x_{01}, x_{02} and x_{12}. The next level simplices are tetrahedrons. The indices are always increasing. The direction of the edges of the triangle is fixed by the indices so that they are increasing. The d_i functions are called face maps as they give the faces of a simplex. The faces of an edge are two points, the faces of a triangle are three edges, the faces of a tetrahedron are triangles. In general, an n-dimensional simplex has $n+1$ $(n-1)$-dimensional simplices as it's faces given by the $n+1$ face maps $d_i : X[n] \to X[n-1]$ for $i = 0, \dots, n$.

A semisimplicial set X can be thought of as a set of points $X[0]$, a set of lines $X[1]$ for any two points, a set of triangles, tetrahedrons, 4-dimensional simplices, 5-dimensional simplices etc. $X[n]$ is the set of n-dimensional simplices.

3.3.2 Semantic universe

In the semisimplicial set model, contexts will be defined as semisimplicial sets. We define a special context which is the (semantic) universe of types. Then, a type over Γ can be interpreted by a context morphism from Γ to the universe. The semantic universe is defined by the following semisimplicial set:

Definition 3.6 (Semantic universe). *We define a functor* $\mathsf{W} : \Delta_+^{op} \to \mathbf{Set}$ *by the following data.* $Hom_{\Delta_+}([m],[n])$ *is written as* $[m] \to [n]$.

- *object part:*

$$\begin{aligned}\mathsf{W}[n] :\equiv \big(&P : {}_{([m]:|\Delta_+|)} \to ([m] \to [n]) \to \mathbf{Set}\big)\\ \times \big(& \cdot\langle\cdot\rangle : {}_{([m]:|\Delta_+|)(f:[m]\to[n])}\\ &\to P(f) \to_{([p]:|\Delta_+|)}\to (g : [p] \to [m]) \to P(f \circ g)\big)\\ \times \big(&{}_{([m]:|\Delta_+|)(f:[m]\to[n])}(u : P(f)) \to u\langle 1\rangle = u\big)\\ \times \big(&{}_{([m],[p],[o]:|\Delta_+|)(f:[m]\to[n])}\\ &(g : [p] \to [m])(h : [o] \to [p])(u : P(f))\\ &\to u\langle g \circ h\rangle = (u\langle g\rangle)\langle h\rangle\big)\end{aligned}$$

- *for the morphism part, given an* $f' : [n] \to [n']$*, we define a function* $\mathsf{W}f' : \mathsf{W}[n'] \to \mathsf{W}[n]$ *by* $\mathsf{W}f'(P) \equiv P(f' \circ \cdot\,)$*. More precisely:*

$$\begin{aligned}\mathsf{W}f'(P, \cdot\langle\cdot\rangle,\ &idlaw, complaw) :\equiv\\ &\big(\lambda_{\,m}\, f\,.\, P({}_m,\ f' \circ f),\\ &\lambda_{\,[m]\ [f]}\, u_{\,[p]}\, g\,.\ \cdot\langle\cdot\rangle({}_{[m]},\ {}_{f'\circ f},\ u,\ {}_{[p]},\ g),\\ &\lambda_{\,[m]\ f}\, u\,.\, idlaw({}_{[m]},\ {}_{f'\circ f},\ u),\\ &\lambda_{\,[m]\ [p]\ [o]\ f}\, g\, h\, u\,.\, complaw({}_{[m]},\ {}_{[p]},\ {}_{[o]},\ {}_{f'\circ f},\ g, h, u)\big)\end{aligned}$$

The functor laws hold. $\mathsf{W}[n]$ is a set of families indexed by morphisms with target $[n]$ (hence their domain is some $[m]$ where $m < n$) equipped with an operation $\cdot\langle\cdot\rangle$ obeying functor-like laws.

We describe a notation for W. We write $P : \mathsf{W}[n]$ for $(P, \cdot\langle\cdot\rangle, idlaw, complaw) : \mathsf{W}[n]$. Morphisms in the category Δ_+ (strictly increasing injections) are isomorphic to finite sets of natural numbers: a set represents the function of which it is the image of. For example the $[2] \to [4]$ function

$$\begin{array}{r}0\\ 0 \mapsto 1\\ 2\\ 1 \mapsto 3\\ 2 \mapsto 4\end{array}$$

can be represented by the set $\{1, 3, 4\}$ which we write as 134. If $I \subset \mathbb{N}$ finite, we define $\mathsf{W}(I)$ to be $\mathsf{W}[|I| - 1]$. If $J \subseteq I$ and $P : \mathsf{W}(I)$ we write $P(J)$ for $P(f \circ \cdot\,) : \mathsf{W}(J)$ where $f : [|J| - 1] \to [|I| - 1]$ is the injection represented by J. We write $P(J)$ also for $P(f) \equiv P(J)(1) : \mathbf{Set}$. If $u : P(I) : \mathsf{W}(I)$[10], then we write $u(J)$ for $u\langle f\rangle : P(J) : \mathsf{W}(J)$.

If $I \subset \mathbb{N}$ finite, $a : I$, $u : P : W(I)$, we write $\partial_a u : \partial_a P : W(I - a)$ for $u(I - a) : P(I - a) : W(I - a)$.

[10]This is a short notation for $u : P(I)$ where $P(I) \equiv P(I)(1)$ and for $P(I) : \mathsf{W}(I)$ at the same time.

$\mathsf{W}(0) = \mathbf{Set}$ is the set of types, an element of $\mathsf{W}(0)$ is a type. An element $P : \mathsf{W}(01)$ is a heterogeneous binary relation or an edge:

$P \equiv P(01) : \mathsf{W}(01)$	set of pairs
$P(0) : \mathsf{W}(0)$	domain type
$P(1) : \mathsf{W}(1) \equiv \mathsf{W}(0)$	target type
$u : P(01) \mapsto u(0) : P(0)$	first projection
$u : P(01) \mapsto u(1) : P(1)$	second projection

An element $P : \mathsf{W}[2]$ is a triangle:

$P \equiv P(012) : \mathsf{W}(012)$	set of triangles
$P(0), P(1), P(2) : \mathsf{W}(0)$	sets of vertices
$P(01), P(02), P(12) : \mathsf{W}(01)$	sets of edges
$u(0) : P(0), u(1) : P(1), u(2) : P(2)$	vertices of the triangle $u : P(012)$
$u(01) : P(01), u(02) : P(02), u(12) : P(12)$	edges of the triangle $u : P(012)$

We can express the definition of W also in type theory just as we did for semisimplicial sets:

$$\mathsf{W}[0] : \mathbf{Set}$$
$$\mathsf{W}[0] = \mathbf{Set}$$
$$\mathsf{W}[1] : \mathsf{W}[0] \to \mathsf{W}[0] \to \mathbf{Set} = \mathbf{Set} \to \mathbf{Set} \to \mathbf{Set}$$
$$\mathsf{W}[1] = \lambda P(0)\, P(1)\, .\, P(0) \to P(1) \to \mathbf{Set}$$
$$\begin{aligned}\mathsf{W}[2] :\ & {}_{(P(0)\,P(1)\,P(2)\,:\,\mathsf{W}[0])} \to \mathsf{W}[1]\, P(0)\, P(1) \to \mathsf{W}[1]\, P(0)\, P(2) \\ & \to \mathsf{W}[1]\, P(1)\, P(2) \to \mathbf{Set} \\ & =_{(P(0)\,P(1)\,P(2)\,:\,\mathbf{Set})} \to (P(0) \to P(1) \to \mathbf{Set}) \to (P(0) \to P(2) \to \mathbf{Set}) \\ & \to (P(1) \to P(2) \to \mathbf{Set}) \to \mathbf{Set}\end{aligned}$$
$$\begin{aligned}\mathsf{W}[2] = \lambda\, & {}_{P(0)\,P(1)\,P(2)}\, P(01)\, P(02)\, P(12)\, .\, {}_{(p_0 : P(0))(p_1 : P(1))(p_2 : P(2))} \to P(01)\, p_0\, p_1 \\ & \to P(02)\, p_0\, p_2 \to P(12)\, p_1\, p_2 \to \mathbf{Set}\end{aligned}$$

An element $P : \mathsf{W}[n]$ is the set of heterogeneous n-dimensional simplices. So, if $P : \mathsf{W}[2]$ and X is a semisimplicial set, P is like $X[2]$ which is a set of triangles, but unlike $X[2]$, the triangles in P are heterogeneous, that is, there are 3 different types for vertices ($P(0)$, $P(1)$, $P(2)$) and also three different types for the edges ($P(01)$, $P(02)$, $P(12)$, these are all parameterised over the appropriate types of vertices). For a triangle in $X[2]$, all vertices have type $X[0]$ and all edges have type $X[1]$. In general, if $P : \mathsf{W}[n]$, P consists of $n+1$ sets, $\binom{n+1}{2}$ types of edges, $\binom{n+1}{3}$ types of triangles, ..., $\binom{n+1}{i}$ types of i-simplices, ..., $n+1$ types of $(n-1)$-simplices, and it is itself a type of n-simplices.

An element $u : P : \mathsf{W}(I)$ is a heterogeneous n-dimensional simplex. It's $(n-1)$-dimensional faces are $\partial_a u : \partial_a P$ for $a : I$. Note that $u : P(I - a)$ and $v : P(I - b)$ are *compatible* (they have a common face) if $\partial_b u = \partial_a v : P(I - ab)$. If $u : P(I)$ and $a, b : I$ distinct then $\partial_a u : P(I - a)$ is compatible with $\partial_b u : P(I - b)$ since $\partial_b \partial_a u = \partial_a \partial_b u : P(I - ab)$.

3.3.3 CwF definition

Definition 3.7 (Semisimplicial set model)**.** *We give the following CwF:*

- *$\mathcal{C}$ is defined as the functor category* $\mathbf{Set}^{\Delta_+^{op}}$. *If* $\Gamma : \Delta_+^{op} \to \mathbf{Set}$ *functor,* $f : [m] \to [n], \rho : \Gamma[n]$, *we write* $\rho f : \Gamma[m]$ *for the operation on morphisms. A substitution* $\sigma : \Delta \to \Gamma$ *is a natural transformation i.e.* $\sigma[n] : \Delta[n] \to \Gamma[n]$ *s.t. for all* $\rho : \Delta[n]$ *we have that* $(\sigma[n](\rho))f = \sigma[m](\rho f)$.

- *Given a* $\Gamma : |\Delta_+^{op} \to \mathbf{Set}|$, *types over* Γ *are defined as natural transformations (substitutions) from* Γ *to* W*:*

$$\begin{aligned} Ty(\Gamma) :\equiv\ & \big(A : {}_{([n]:|\Delta_+|)} \to \Gamma[n] \to \mathsf{W}[n]\big) \\ & \times \big({}_{([n]:|\Delta_+|)}(\rho : \Gamma[n])(f : [m] \to [n]) \to A({}_{[n]},\, \rho)(f \circ \cdot\,) = A({}_{[m]},\, \rho f)\big) \end{aligned}$$

  The naturality property could have been expressed by $A(\rho)(f) = A(\rho f)(1)$. From such a rule the pointwise equality of the families $A({}_{[n]},\, \rho)(f \circ \cdot\,)$ and $A({}_{[m]},\, \rho f)$ in $\mathsf{W}[n]$ can be derived.

  If $A : Ty(\Gamma)$ *and* $\sigma : \Delta \to \Gamma$ *then* $A\sigma :\equiv \lambda_{\,[n]}\, \rho\, .\, A({}_{[n]},\, \sigma[n](\rho))$ *(type substitution is composition of natural transformations).*

- *Given a* $\Gamma : |\Delta_+^{op} \to \mathbf{Set}|$ *and an* $A : Ty(\Gamma)$, *terms of type* A *are defined as follows:*

$$\begin{aligned} Tm(\Gamma, A) :\equiv\ & \big(t : {}_{([n]:|\Delta_+|)}(\rho : \Gamma[n]) \to A({}_{[n]},\, \rho)(1)\big) \\ & \times \big({}_{([m][n]:|\Delta_+|)}(f : Hom_{\Delta_+}([m],[n]))(\rho : \Gamma[n]) \\ & \quad \to \underbrace{\underbrace{t(\rho)}_{:A(\rho)(1)}\ \langle f \rangle}_{:A(\rho)(f)} = \underbrace{t(\ \underbrace{\rho f}_{:\Gamma[m]}\ )}_{:A(\rho f)(1)}\ \big) \end{aligned}$$

  Note that the equation $t(\rho)\langle f \rangle = t(\rho f)$ is well typed because of the naturality of Ty.

  If $t : Tm(\Gamma, A)$ *and* $\sigma : \Delta \to \Gamma$ *then*

$$\begin{aligned} t\sigma &: {}_{([n]:|\Delta_+|)}(\rho : \Delta[n]) \to A({}_{[n]},\, \sigma[n](\rho))(1) \\ t\sigma &:\equiv \lambda_{\,[n]}\, \rho\, .\, t({}_{[n]},\, \sigma[n](\rho)) \end{aligned}$$

- *Given* $A : Ty(\Gamma)$, *the comprehension* $\Gamma.A$ *is the* $\Delta_+^{op} \to \mathbf{Set}$ *functor defined by:*
  - $(\Gamma.A)[n] :\equiv (\rho : \Gamma[n]) \times A({}_{[n]},\, \rho)(1)$
  - *if* $f : [m] \to [n]$ *then* $(\rho, u)f :\equiv (\rho f, u\langle f \rangle)$

A context is a semisimplicial set, that is, a set of points (elements of the context), for each two point, a set of edges (expressing some relation between the two points), a set of triangles (expressing relations between relations) etc.

Before looking at what a type over Γ is, we need to fix a level n. An open type at level n is a mapping from n-simplices in $\Gamma[n]$ to a set of heterogeneous n-simplices. A closed type is an element of $\mathsf{W}[n]$, that is, a set of heterogeneous n-simplices.

At level n, an open term is a mapping from n-simplices to a heterogeneous n-simplex of the corresponding type. A closed term is a heterogeneous n-simplex of the corresponding closed type.

3.3.4 Function types

Given $A : Ty(\Gamma), B : Ty(\Gamma.A)$, $\Pi\ A\ B : Ty(\Gamma)$ is defined as follows. $\Pi\ A\ B$ is a natural transformation from Γ to W i.e. $\Pi\ A\ B :\ {}_{([n]:|\Delta_+|)} \to \Gamma[n] \to \mathsf{W}[n]$ together with a naturality property. We fix an $[n]$ and a $\rho : \Gamma[n]$ and provide an element of $\mathsf{W}[n]$. Afterwards, we will show the naturality property.

We define some abbreviations:

$$\begin{aligned}
A' &:\equiv A({}_{[n]},\, \rho) : \mathsf{W}[n] \\
B' &:\equiv \lambda\,{}_{([m]:|\Delta_+|)}\,(f : [m] \to [n])(a : A'(f)).\, \overbrace{B({}_{[m]}, (\rho f, a))}^{:\mathsf{W}[m]}
\end{aligned}$$

Definition 3.8 (Function types in the semisimplicial set model)**.** $(\Pi AB)({}_{[n]},\, \rho) :$ $\mathsf{W}[n]$ *is defined as the following quadruple:*

$$\begin{aligned}
&\Big(P :\equiv \lambda\,{}_{[n']}\,(f' : [n'] \to [n])\,. \\
&\qquad\quad \big(\varphi :\ {}_{([m]:|\Delta_+|)}\,(f : [m] \to [n'])(a : A'(f' \circ f)) \to B'(f' \circ f, a)(1)\big) \\
&\qquad\quad \times \big({}_{([m]:|\Delta_+|)(f:[m]\to[n'])}\,(a : A'(f' \circ f))\,{}_{([p]:|\Delta_+|)}\,(g : [p] \to [m]) \\
&\qquad\qquad \to \varphi(f \circ g, a\langle g\rangle) = \varphi(f, a)\langle g\rangle\big) \\
&, \cdot\langle\cdot\rangle :\equiv \lambda\,{}_{[n'](f':[n']\to[n])}\,((\varphi, \tau) : P(f'))\,{}_{[n'']}\,(f'' : [n''] \to [n'])\,. \\
&\qquad\quad \big(\lambda\,{}_{[m]}\,(f : [m] \to [n''])(a : A'(f' \circ f'' \circ f))\,.\,\varphi({}_{[m]},\, f'' \circ f, a) \\
&\qquad\quad , \lambda\,{}_{[m](f:[m]\to[n''])}\,(a : A'(f' \circ f'' \circ f))\,{}_{[p]}\,(g : [p] \to [m])\,.\,\tau({}_{[m]},\ {}_{f''\circ f},\ a, g)\big) \\
&, \qquad \lambda\,{}_{[n'](f':[n']\to[n])}\,((\varphi, \tau) : P(f'))\,. \\
&\qquad\quad \textit{a proof of } (\varphi, \tau)\langle 1\rangle = (\varphi, \tau) \\
&\qquad\quad \textit{which equals } (\varphi(1 \circ \cdot, \cdot) = \varphi(\cdot, \cdot)) \times (\tau(.,\ {}_{1\circ .},\ \cdot, \cdot) = \tau(.,\ .,\ \cdot, \cdot)) \\
&\qquad\quad \textit{which equals } (\varphi(\cdot, \cdot) = \varphi(\cdot, \cdot)) \times (\tau(.,\ .,\ \cdot, \cdot) = \tau(.,\ .,\ \cdot, \cdot)) \\
&, \qquad \lambda\,{}_{[n'][n''][n'''](f':[n']\to[n])}(f'' : [n''] \to [n'])(f''' : [n'''] \to [n''])((\varphi, \tau) : P(f'))\,. \\
&\qquad\quad \textit{a proof of } (\varphi, \tau)\langle f'' \circ f'''\rangle = ((\varphi, \tau)\langle f''\rangle)\langle f'''\rangle \\
&\qquad\quad \textit{which equals } (\varphi(f'' \circ f''' \circ \cdot, \cdot) = \varphi(f'' \circ \cdot, \cdot)\langle f'''\rangle) \\
&\qquad\qquad\qquad\qquad \times (\tau(.,\ {}_{f''\circ f'''\circ .},\ \cdot, \cdot) = \tau(.,\ {}_{f''\circ .},\ \cdot, \cdot)\langle f'''\rangle) \\
&\qquad\quad \textit{which equals } (\varphi(f'' \circ f''' \circ \cdot, \cdot) = \varphi(f'' \circ f''' \circ \cdot, \cdot)) \\
&\qquad\qquad\qquad\qquad \times (\tau(.,\ {}_{f''\circ f'''\circ .},\ \cdot, \cdot) = \tau(.,\ {}_{f''\circ f'''\circ .},\ \cdot, \cdot))\Big)
\end{aligned}$$

In short, a function $\varphi : (\Pi\ A\ B)({}_{[n]},\, \rho)(1)$ is defined as a mapping from an $a : A'(f)$ to an element of $B'(f, a)(1)$ for all $f : [m] \to [n]$, which commutes with applying $\cdot\langle g\rangle$.

Naturality means that for all $g : [p] \to [n]$ and $\rho : \Gamma[n]$ we have $(\Pi AB)({}_{[n]},\, \rho)(g \circ \cdot\,) = (\Pi\, A\, B)({}_{[p]},\, \rho g)$. A and B are natural, i.e. $A({}_{[n]},\, \rho)(g \circ \cdot\,) = A({}_{[p]},\, \rho g)$ and similarly for B, so by the congruence rules of equality we know $\Pi\ A\ B$ is also natural.

3.4 Kan semisimplicial sets

A Kan semisimplicial set is a semisimplicial set (a functor $\Delta_+^{op} \to \mathbf{Set}$) equipped with additional operations called *completion* and *filling*. Having a completion operator means that e.g. at level 2, given two compatible edges we get a third edge which completes the two edges so that they can form a triangle (only the third edge is produced by the completion operator). Similarly, given 3 triangles, the completion on level 3 produces a fourth triangle so that the 4 triangles together have the shape of a tetrahedron. Filling lets one build simplices on higher levels. Filling on level 2 means that given two compatible edges, together with the third edge produced by the completion operation, these form a triangle: filling produces the triangle, an object on level 3.

Definition 3.9 (Set of compatible families)**.** *Given a $J \subset \mathbb{N}$ finite, $a : J$, the set of compatible families $P(\Lambda^a(J))$ is defined as the set of families $u_b : P(J-b)$ for $b : J - a$ such that they are compatible on all faces i.e. $\partial_c u_b = \partial_b u_c : P(J - bc)$ for each distinct $b, c : J - a$.*

A compatible family u : $P(\Lambda^a(J)) \equiv (b : J - a) \to P(J - b)$ is a $|J| - 1$-dimensional horn almost forming a $|J| - 1$ dimensional simplex. Only one face of the simplex is left out, the one numbered by a. For example, if $J = 012$ and $a = 1$, the family u_0, u_2 are two edges which connect at vertex number 1.

There is a canonical map $P(J) \to P(\Lambda^a(J))$ which maps a $u : P(J)$ into the family $\partial_b u : P(J - b)$ for $b : J - a$.

Definition 3.10 (V)**.** *A $P : W(I)$ is in $V(I)$ if for all $J \subseteq I$, each canonical map $P(J) \to P(\Lambda^a(J))$ has a section.*

This means that if we have a horn, we can fill it and get a simplex which is one dimension higher and this simplex has the same faces as the original horn.

Definition 3.11 (Kan semisimplicial set)**.** *Given a semisimplicial set X, for all $I \subset \mathbb{N}$ finite the restriction $X(I) \equiv X[|I| - 1]$ can be regarded as an element of $\mathsf{W}(I)$. X is called a Kan semisimplicial set if each restriction $X(I)$ is in $\mathsf{V}(I)$.*

So a semisimplicial set X is Kan if for any level n, any n-dimensional horn can be filled to form an n-dimensional simplex.

3.5 Implementation details

The formalisation of categories with families and the term model for simple type theory is a work in progress in Agda. It uses function extensionality, higher inductive types and induction-induction [6] in an essential way, however all the types are sets. The metatheory required is essentially a version of observation type theory [5] extended with higher inductive types and propositional proof irrelevance. However, it is possible to do such a formalisation in Agda even without having definitional computation rules for function extensionality and without having pattern matching for higher inductive types, it is just more cumbersome.

3.5.1 Simple type theory in 0-level homotopy type theory

I implemented the term model of simple type theory in type theory.

The Agda implementation first defines the simplified CwF model (section 3.1) as an interface that one could implement: a record type having as fields the category of contexts and the contravariant functor from the contexts to types and terms together with the definition of function space. (It is simplified for simple type theory: the `Ty` part of the functor is just a constant functor.)

```
record STT : Set1 where
  field
    -- category of contexts
    Con    : Set
    _⇒_    : Con → Con → Set
    one    : {Γ : Con} → Γ ⇒ Γ
    _○_    : {Δ Γ Θ : Con} → Γ ⇒ Θ → Δ ⇒ Γ → Δ ⇒ Θ
    lid    : {Δ Γ : Con} (σ : Δ ⇒ Γ) → one ○ σ = σ
    rid    : {Δ Γ : Con} (σ : Δ ⇒ Γ) → σ ○ one = σ
    ○-ass  : {Δ Γ Θ Ω : Con} (σ : Θ ⇒ Ω) (δ : Γ ⇒ Θ) (ν : Δ ⇒ Γ)
           → (σ ○ δ) ○ ν = σ ○ (δ ○ ν)
    -- Con has a terminal object
    ◇      : Con
    !      : (Γ : Con) → Γ ⇒ ◇ --?
    !uniq  : {Γ : Con} → (δ : Γ ⇒ ◇) → δ = ! Γ

    -- set of types
    Ty     : Set

    -- a contravariant functor Tm_A : Con → Set for all (A : Ty)
    Tm      : Ty → Con → Set
    _[_]    : {A : Ty} {Δ Γ : Con} → Tm A Γ → Δ ⇒ Γ → Tm A Δ
    []id    : {A : Ty} {Γ : Con} (t : Tm A Γ) → t [ one ] = t
    []cong  : {A : Ty} {Δ Γ Θ : Con} (t : Tm A Θ) (σ : Γ ⇒ Θ)
             (δ : Δ ⇒ Γ) → t [ σ ○ δ ] = (t [ σ ]) [ δ ]

    -- comprehension
    _•_    : (Γ : Con) (A : Ty) → Con
    p      : {Γ : Con} {A : Ty} → Γ • A ⇒ Γ
    q      : {Γ : Con} {A : Ty} → Tm A (Γ • A)
    ⟨_,_⟩   : {Δ Γ : Con} {A : Ty} (σ : Δ ⇒ Γ) (u : Tm A Δ)
            → Δ ⇒ Γ • A
    pβ     : {Δ Γ : Con} {A : Ty} (σ : Δ ⇒ Γ) (u : Tm A Δ)
            → p ○ ⟨ σ , u ⟩  = σ
    qβ     : {Δ Γ : Con} {A : Ty} (σ : Δ ⇒ Γ) (u : Tm A Δ)
            → q [ ⟨ σ , u ⟩ ] = u
    pqη    : {Δ Γ : Con} {A : Ty} (δ : Δ ⇒ Γ • A)
            → ⟨ p ○ δ , q [ δ ] ⟩ = δ

    -- function space: Tm B (Γ • A) ≃ Tm (A ⇒ B) Γ
    _⇒_ : Ty → Ty → Ty
    lam  : {Γ : Con} {A B : Ty} → Tm B (Γ • A) → Tm (A ⇒ B) Γ
    app  : {Γ : Con} {A B : Ty} → Tm (A ⇒ B) Γ → Tm B (Γ • A)
    ⇒β   : {Γ : Con} {A B : Ty} (t : Tm B (Γ • A))
          → app (lam t) = t
    ⇒η   : {Γ : Con} {A B : Ty} (f : Tm (A ⇒ B) Γ)
          → lam (app f) = f
    lamnat : {Δ Γ : Con} {A B : Ty} (δ : Δ ⇒ Γ) (u : Tm B (Γ • A))
             → lam (u [ ⟨ δ ○ p , q ⟩ ]) = (lam u) [ δ ]
```

In the future we might also need to include in the definition that Con, Ty and Term are indeed sets (types of h-level 0).

The term model is given by mutually defined inductive types with higher constructors which we just postulate. The eliminator and computation rules are also given by postulates. There is a trick by Dan Licata [39] to provide definitional computation rules for 0-constructors, we use this in the implementation, however we hide the usage of this trick here because it makes the code harder to read. I list the type formation rules and construtors below:

```
data Ty : Set where
  o    : Ty
  _⇒_ : (A B : Ty) → Ty

data Con : Set where
  ◇    : Con
  _•_ : (Γ : Con) (A : Ty) → Con

data _⇒_ : Con → Con → Set where
  one    : {Γ : Con} → Γ ⇒ Γ
  _∘_    : {Γ Δ Θ : Con} → Δ ⇒ Θ → Γ ⇒ Δ → Γ ⇒ Θ
  p      : {Γ : Con} {A : Ty} → Γ • A ⇒ Γ
  ⟨_,_⟩  : {Δ Γ : Con} {A : Ty} → Δ ⇒ Γ → Tm A Δ → Δ ⇒ Γ • A

data Tm  : Ty → Con → Set where
  q    : {Γ : Con} {A : Ty} → Tm A (Γ • A)
  _[_] : {Δ Γ : Con} {A : Ty} → Tm A Γ → Δ ⇒ Γ → Tm A Δ
  lam  : {Γ : Con} {A B : Ty} → Tm B (Γ • A) → Tm (A ⇒ B) Γ
  app  : {Γ : Con} {A B : Ty} → Tm (A ⇒ B) Γ → Tm B (Γ • A)

postulate
  -- higher constructors for _⇒_
  lid   : {Δ Γ : Con} (σ : Δ ⇒ Γ) → one ∘ σ = σ
  rid   : {Δ Γ : Con} (σ : Δ ⇒ Γ) → σ ∘ one = σ
  ∘-ass : {Δ Γ Θ Ω : Con} (σ : Θ ⇒ Ω) (δ : Γ ⇒ Θ) (ν : Δ ⇒ Γ)
          → (σ ∘ δ) ∘ ν = σ ∘ (δ ∘ ν)
  pβ    : {Δ Γ : Con} {A : Ty} (σ : Δ ⇒ Γ) (u : Tm A Δ)
          → p ∘ ⟨ σ , u ⟩ = σ
  pqη   : {Δ Γ : Con} {A : Ty} (δ : Δ ⇒ Γ • A)
          → ⟨ p ∘ δ , q [ δ ] ⟩ = δ
  ⇒set : {Δ Γ : Con} {σ δ : Δ ⇒ Γ} (p q : σ = δ) → p = q

  -- higher constructors for Tm
  []id   : {A : Ty} {Γ : Con} (t : Tm A Γ) → t [ one ] = t
  []cong : {A : Ty} {Δ Γ Θ : Con} (t : Tm A Θ) (σ : Γ ⇒ Θ)
           (δ : Δ ⇒ Γ) → t [ σ ∘ δ ] = (t [ σ ]) [ δ ]
  qβ     : {Δ Γ : Con} {A : Ty} (σ : Δ ⇒ Γ) (u : Tm A Δ)
           → q [ ⟨ σ , u ⟩ ] = u
  ⇒β     : {Γ : Con} {A B : Ty} (t : Tm B (Γ • A))
           → app (lam t) = t
  ⇒η     : {Γ : Con} {A B : Ty} (f : Tm (A ⇒ B) Γ)
```

```
          → lam (app f) = f
  lamnat : {Δ Γ : Con} {A B : Ty} (δ : Δ ⇒ Γ) (u : Tm B (Γ • A))
          → lam (u [ ⟨ δ ∘ p , q ⟩ ]) = (lam u) [ δ ]
  Tmset  : {Γ : Con} {A : Ty} {u v : Tm A Γ} (p q : u = v)
          → u = v
```

The elimination rules provide definitional computation rules for the normal constructors and we postulate propositional computation rules for the higher constructors. For example, the type of the eliminator for terms looks as follows:

```
Tm-elim :
  (P       : (A : Ty) (Γ : Con) → Tm A Γ → Set)
  (cq      : {Γ : Con} {A : Ty} → P A (Γ • A) q)
  (c[]     : {Δ Γ : Con} {A : Ty} {u : Tm A Γ} (s : P A Γ u)
             (δ : Δ ⇒' Γ) → P A Δ (u [ δ ]))
  (clam    : {Γ : Con} {A B : Ty} {u : Tm B (Γ • A)}
             (s : P B (Γ • A) u) → P (A ⇒ B) Γ (lam u))
  (capp    : {Γ : Con} {A B : Ty} {f : Tm (A ⇒ B) Γ}
             (s : P (A ⇒ B) Γ f) → P B (Γ • A) (app f))
  (c[]id   : {A : Ty} {Γ : Con}  {t : Tm A Γ} (s : P A Γ t)
             → P A Γ : c[] s one ≡[ []id t ] s)
  (c[]cong : {A : Ty} {Δ Γ Θ : Con} {t : Tm A Θ} {σ : Γ ⇒' Θ}
             {δ : Δ ⇒' Γ} (s : P A Θ t)
             → P A Δ : c[] s (σ ∘ δ) ≡[ []cong t σ δ ] c[] (c[] s σ) δ)
  (cqβ     : {Δ Γ : Con} {A : Ty} (σ : Δ ⇒' Γ) {u : Tm A Δ}
             (s : P A Δ u) → P A Δ : c[] cq ⟨ σ , u ⟩ ≡[ qβ σ u ] s)
  (c⇒β      : {Γ : Con} {A B : Ty} {t : Tm B (Γ • A)}
             (s : P B (Γ • A) t)
             → P B (Γ • A) :capp (clam s) ≡[ ⇒β t ] s)
  (c⇒η      : {Γ : Con} {A B : Ty} {f : Tm (A ⇒ B) Γ}
             (s : P (A ⇒ B) Γ f)
             → P (A ⇒ B) Γ : clam (capp s) ≡[ ⇒η f ] s)
  (clamnat : {Δ Γ : Con} {A B : Ty} (δ : Δ ⇒' Γ) {u : Tm B (Γ • A)}
             (s : P B (Γ • A) u)
             → P (A ⇒ B) Δ : clam (c[] s ⟨ δ ∘ p , q ⟩)
               ≡[ lamnat δ u ] c[] (clam s) δ )
  (cset    : {A : Ty} {Γ : Con} {u v : Tm A Γ} (p q : u ≡ v)
             (a : P A Γ u) (b : P A Γ v) (r : P A Γ : a ≡[ p ] b)
             (s : P A Γ : a ≡[ q ] b)
             → (λ z → P A Γ : a ≡[ z ] b) : r ≡[ Tmset p q ] s)
  {A : Ty} {Γ : Con} (x : Tm A Γ) → P A Γ x
```

4 Future plans

Metaprogramming for a programming language is an important tool for abstraction. To do dependently typed metaprogamming it is an essential requirement to internalise the syntax of type theory in type theory (which means defining the term model within type theory). Efforts to achieve this have resulted in incomplete and notationally very heavy constructions [20] [15]. Another more semantic approach is using observational type theory [47].

Defining a model of type theory amounts to defining a function from the syntax (term model) to a semantic domain for which the internalisation of the syntax is also required. Hence, this is a step required for solving the canonicity problem of homotopy type theory by model construction.

A major difficulty in internalising the syntax is the treatment of definitional equalities. They can be given by separate constructors for each type which amounts to defining setoids as opposed to sets but then one ends up with many coherence laws inducing further coherence laws etc. An idea which comes from homotopy type theory is the usage of higher inductive types for the same purpose. In this case, one gets all the coherence laws by the $\mathsf{transport}$ and ap functions defined for propositional equality.

On a related note, the fact that propositional equality of definitionally equal terms in the empty context is provable by refl can be used to model the object theory definitional equalities by the definitional equality of the metatheory which is unreachable from within the theory.

For my PhD research, I would like to explore and extend the usability of this technique for defining type theory inside type theory. Also, I would like to understand what features a type theory should have to allow such constructions and to implement such a type theory with normalisation as a shallow embedding in Agda (this would be a 0-truncated version of homotopy type theory, or an extension of observation type theory with higher inductive types, induction-induction, induction-recursion using a propositionally proof-irrelevant universe of propositions).

4.1 Thesis plan

An imaginary plan for the structure of my thesis is as follows.

Title: Formalising type theory in type theory.

Abstract: Derivation rules of a type theory are given by mutual definitions using induction-induction, sometimes induction-recursion. For example the context judgements $\Gamma \vdash$ are interpreted by an inductive type $\mathsf{Con} : \mathsf{U}$ and the types over Γ are interpreted by an inductive type indexed over Con, $\mathsf{Ty} : \mathsf{Con} \to \mathsf{U}$. One of the constructors of Con depends on a Ty: $\cdot,\cdot : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Con}$. This is an example of induction-induction. When defining a universe, one needs to use induction-recursion as described in section 2.1.11. The judgements for definitional equality can be given explicitly as separate inductive datatypes for contexts, types and terms as shown in section 2.1.2. In this case one has to augment all type and term formers with congruence rules in the corresponding inductive type for equality. Another solution is to use an idea from homotopy type theory: types with higher constructors expressing these equalities. In this case, because propositional equality has the property that equal terms can be replaced with each other ($\mathsf{transport}$), we get all congruence and substitution rules for free. However, we would need normalisation for such a type theory to be able to use it to implement models giving a computational explanation to homotopy type theory. And the computational interpretation of higher inductive types is just what we would like to find. Luckily, all the rules define sets, types in homotopy type theory having h-level 0. And for these types the normalisation problem can be solved as shown by observational type theory [5]. My aim is to implement a version of observational type theory with set-level higher inductive types where the universe of propositions is not definitionally, but propositionally proof irrelevant. The theory should support function extensionality and induction-induction, induction-recursion in a principled way without using pattern matching. My imaginary thesis consists of the following chapters:

1. Introduction

2. Technical background. This chapter would have very similar content to section 2 in this document with some additional material mentioned at each subsection.

    1.1. Type theory. In addition to the material presented here I would talk about impredicativity and provide more details about induction-recursion, induction-induction and parametricity.

    1.2. Category theory. In addition to the material presented here I would present the Yoneda lemma and more details about presheaf categories.

    1.3. Homotopy type theory. A more detailed introduction than the current one.

3. Models of type theory

    2.1. Categories with families. I would present rules for more type formers and detailed rules on how to define universes and rules for homotopy type theory. This would be augmented with a formal specification of such a model.

    2.2. Some models. Examining the term model would show what features a type theory should exactly have in order to be capable for defining all rules. Short definition of other models.

4. 0-truncated homotopy type theory. This chapter would present the type theory described above together with an implementation in Agda.

5. Applications. This chapter would provide some applications on how to use the above theory to explain the meaning of the informal usages of constructive metatheories as in [10].

    3.1. Term model

    3.2. NBE for simple type theory

    3.3. NBE for type theory

    3.4. Groupoid model

    3.5. 2-groupoid model

    3.6. Semisimplicial set model

    3.7. Kan semisimplicial set model

5 Other topics

Some other topics that I studied in my first year included the following.

5.1 Free applicative functors

Applicative functors [49] are a generalisation of monads in functional programming. Both allow defining effectful computations in a pure setting, however applicative functors can only express computations with a fixed structure. Free monads are used as a tool for producing embedded languages from any language defined by a functor [61]. Paolo Capriotti extended the idea of defining embedded languages from free monads to free applicative functors. These langauges have less expressive power however they allow the static analysis of these. Together with Paolo Capriotti, we wrote a paper about this construction [14] and we plan to extend the paper with a formalisation in terms of (applicative) containers.

5.2 Parametricity

Some models of type theory validate so-called free theorems. These express a generalisation of the idea that polymorphic functions are natural tranformations. This result was first proved for System F [59] and popularised for programming languages based on this type system [63]. They are a useful tool for reasoning about programs. Free theorems can be only proven metatheoretically, not within the theory. A similar property stated in section 2.3.2 is that all types in U_0 are of h-level 0.

5.3 Total Haskell embedded in Agda

Haskell is a programming language with a type system based on the impredicative System F [28]. It has a pure type system however non-total programs can be written in it directly. Haskell programmers typically try to avoid writing partial programs and like to reason about them as if Haskell was a total language [21]. A way to conveniently reason formally about Haskell total programs is to embed them into a total predicative dependent type theory, like that of Agda [53]. However this presents difficulties when try to use free theorems [59] when reasoning about programs because the obvious postulating of free theorems true in the impredicative System F setting might make the predicative theory incosistent. We encountered this problem when trying to formalise free applicative functors. I am interested in finding ways to avoid this problem.

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM, 1996.

[2] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[3] Thorsten Altenkirch. An open question concerning inductive equality. e-mail message to the edinburgh lego club, 1992.

[4] Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

[5] Thorsten Altenkirch, Conor Mcbride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–58. ACM, 2007.

[6] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In Andrea Corradini and Bartek Klin, editors, *CALCO*, Lecture Notes in Computer Science, 2011.

[7] S. Awodey. *Category Theory*. Oxford Logic Guides. OUP Oxford, 2010.

[8] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. September 2007.

[9] Henk Barendregt and Silvia Ghilezan. Lambda terms for natural deduction, sequent calculus and cut elimination. *J. Funct. Program.*, 10(1):121–134, 2000.

[10] Bruno Barras, Thierry Coquand, and Simon Huber. A generalization of Takeuti-Gandy interpretation. `http://uf-ias-2012.wikispaces.com/file/view/semi.pdf`, 2013.

[11] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[12] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001, volume 2152 of Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2000.

[13] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

[14] Paolo Capriotti and Ambrus Kaposi. Free applicative functors. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014.*, pages 2–30, 2014.

[15] James Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, January 2009.

[16] Thierry Coquand. The paradox of trees in type theory. *BIT*, 32, 1991.

[17] Thierry Coquand. Weak type theory. `http://www.cse.chalmers.se/~coquand/wmltt.pdf`, 2012.

[18] Thierry Coquand. About the setoid model. `http://www.cse.chalmers.se/~coquand/setoid.pdf`, 2013.

[19] Thierry Coquand and Simon Huber. Simplicial sets model of type theory. `http://www.cse.chalmers.se/~coquand/decuniv.pdf`, 2013.

[20] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006.

[21] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 206–217. ACM, 2006.

[22] Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

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

[24] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[25] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Trans. Amer. Math. Soc.*, 58:231–294, 1945.

[26] J. Y. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. 63:63–92, 1971.

[27] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[28] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge University Press, New York, NY, USA, 1989.

[29] Healfdene Goguen, Conor Mcbride, and James Mckinna. Eliminating dependent pattern matching. In *of Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

[30] Robert Harper. *Practical Foundations for Programming Languages.* December 2009.

[31] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[32] M. Hofmann. *Extensional Concepts in Intensional Type Theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.

[33] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[34] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[35] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[36] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations, 2012. arXiv:1211.2851.

[37] Nicolai Kraus and Christian Sattler. On the hierarchy of univalent universes: u_n is not n-truncated. `http://red.cs.nott.ac.uk/~ngk/universes.pdf`, 2013.

[38] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic.* Cambridge University Press, New York, NY, USA, 1986.

[39] Dan Licata. Running circles around (in) your proof assistant; or, quotients that compute, 2011. `http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/`.

[40] Sam Lindley and Conor Mcbride. Hasochism the pleasure and pain of dependently typed haskell programming, 2013. Haskell Symposium 2013.

[41] Saunders Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer, 2nd edition, September 1998.

[42] Lorenzo Malatesta, Thorsten Altenkirch, Neil Ghani, Peter Hancock, and Conor McBride. Small induction recursion, indexed containers and dependent polynomials are equivalent, 2013. TLCA 2013.

[43] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[44] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, 1984.

[45] Nicolas Tabareau Matthieu Sozeau. Univalence for free, 2013. .

[46] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[47] Conor McBride. Dependently typed metaprogramming (in agda). a summer course at the university of cambridge, 2013. .

[48] Conor Mcbride and James Mckinna. I am not a number: I am a free variable. In *In Haskell workshop*, 2004.

[49] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[50] Paul-André Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications, Edinburgh*, volume 902 of *Lecture Notes in Computer Science*, pages 328–334. Springer, 1995.

[51] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* MIT Press, 1997.

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

[53] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[54] Erik Palmgren. On universes in type theory. In *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, page 191 – 204. Oxford University Press, 1998.

[55] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas J ohnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

[56] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* 1991.

[57] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks*, page pages, 1992.

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

[59] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, September 19-23, 1983*, pages 513–523. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1983.

[60] Thomas Streicher. Investigations into intensional type theory. habilitation thesis, 1993. .

[61] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

[62] Vladimir Voevodsky. A very short note on the homotopy λ-calculus. `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf`, 2006.

[63] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[64] Philip Wadler. Monads for functional programming, 1995.

[65] Michael A. Warren. The strict ω-groupoid interpretation of type theory. Hart, Bradd (ed.) et al., Models, logics, and higher-dimensional categories: A tribute to the work of Mihály Makkai. Proceedings of a conference, CRM, Montréal, Canada, June 18–20, 2009. Providence, RI: American Mathematical Society (AMS). CRM Proceedings and Lecture Notes 53, 291-340 (2011)., 2011.