Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

Formalization of Type Theory in Agda

Bachelorarbeit

im Studiengang Informatik

von

Maxim Urschumzew

Hannover, 13. August 2018

Contents

1 Introduction

Type theory was first introduced by Russell in 1903 [13] as part of Russell and Whitehead's efforts of writing the *Principia Mathematica*, where its role was to serve as a safe, but still powerful basis for formalizing mathematics - escaping the paradoxes of set theory, such as, "Does the set of all sets, which do not contain themselves, contain itself?" (Russell's paradox). [9]

It is closely related to intuitionistic (constructive) mathematics, a branch of mathematics where proofs are meant to be constructive. Here, a proof of the existence of some object must give us a way to explicitly construct such an object. In practice, this means that axioms which allow us to circumvent explicit constructions, like the principle of excluded middle, are rejected. In the early 1930s, Brouwer, Heyting and Kolmogorov gave a computational interpretation of intuitionistic logic. The discovered principle is known as "Propositions as Types" or "Curry-Howard Isomorphism", the idea is to interpret a proposition as a type, and a proof of it as an algorithm, stated in the lambda calculus, having this type.

Considering the computational aspect as being crucial, Bishop developed analysis in a constructive setting (1967). Inspired by this, different type theories were developed to provide a system in which Bishop's mathematics could be formalized. These type theories form the basis of modern proof assistants, including Agda (Intuitionistic Type Theory, [15]) and Coq (Calculus of Inductive Constructions, [10]). [9]

In 1945, Eilenberg and MacLane introduced category theory, first as a tool for applying algebraic methods to a topological problem. Over time, it developed into a language which could be used to describe the objects of many different branches of mathematics and faciliated the discovery of connections between them. Starting in the 1960s, Lawvere and others explored the idea of applying category theory to the basis of math itself: logic and set theory. This endeavour resulted in the definition of a topos, a special kind of category, which has an internal logic rich enough to serve as a generalized foundational framework. [14]

Furthermore, a direct connection to type theory became apparent, where categories can be seen as providing the semantics for type theories. Thus summarizing, the relationships between mathematics, type theory and category theory can be stated as follows [25]:

$$\begin{tikzcd} & \text{mathematics} \arrow[dl, "\textit{can be formalized in a}"'] \arrow[dr, "\textit{can be internalized in a}"] & \\ \text{type theory (syntax)} \arrow[rr, "\textit{can be interpreted in a}"'] & & \text{category (semantics)} \end{tikzcd}$$

As part of research by Awodey, Warren and Voevodsky around 2006, Homotopy Type Theory (HoTT) was developed. In homotopy theory (without -*Type*-), topological spaces are studied with respect to what paths can be constructed. This includes an infinite hierarchy of paths: between points, between paths between points, between paths between paths between points, and so on. Thus HoTT, being a type theory which has an interpretation in Kan simplicial sets (a category studied in homotopy theory), mirrors these features and offers new ways for doing mathematics in it [27]:

1. A proof of equality $a = b$ is interpreted as a path between a and b. In homotopical fashion, such a proof is thus no longer unique: there may be different paths between a and b. And since such paths may be compared again and again, an infinite path structure emerges.

2. HoTT contains a new axiom: the univalence axiom. It says that isomorphic structures may be treated as being equal:

   $$(A \simeq B) \simeq (A = B)$$

   Such a statement is not consistent with set theory, but it can be assumed in HoTT, allowing us to treat isomorphic structures more intuitively.

Nevertheless, the univalence axiom is only an axiom in HoTT, i.e., it has no computational meaning - code which uses it cannot be executed. This lead to the development of Cubical Type Theory (CTT), which, by modelling equality *explicitly* by paths, succeeded in giving computational meaning to the univalence axiom [8].

While the development is still in progress (for example, CTT currently has no interpretation in as general a class of categories as HoTT has [24]), the state of research can be visualized as follows [25]:

$$\begin{tikzcd} & \text{homotopical mathematics} \arrow[dl, "\textit{can be formalized in}"'] \arrow[dr, "\textit{can be internalized in}"] & \\ \text{HoTT / CTT} \arrow[rr, "\textit{can be interpreted in}"'] & & \text{higher categories} \end{tikzcd}$$

The goal of this thesis is the exploration of the relationship between type theory and category theory, albeit on a much smaller scale: Our topic is the simply typed lambda calculus and its interpretation into a cartesian closed category (CCC):

$$\begin{tikzcd}
& \text{(Propositional Logic)} \arrow[dl, dashed] \arrow[dr, dashed] & \\
\text{ST. } \lambda\text{-Calc} \arrow[rr, "\textit{can be interpreted in a}"'] & & \text{CCC}
\end{tikzcd}$$

In order to present this connection, we formalize it in Cubical Type Theory, as implemented in the Agda proof assistant. This allows us to take advantage of a computing functional extensionality (a corrollary of univalence, does not compute in other type theories). Because of this, some practical aspects of formalizing mathematics in a (cubical) type theory are also touched upon in this thesis.

Motivation

My motivation comes from the praxis of software development. Sometimes, when programming, I would like to have a tool which could provide a higher level view on code: To see the whole structure of a program at once, to see which components exist and how they interact. Especially, this would make it easier to explore and navigate large codebases.

But it should be more than a mere visualization of the code. It should be an interactive representation equivalent to it, such that it would become possible to program on a higher level - graphically managing the connections between lower level components. Also it should be possible to zoom in and out arbitrarily, enabling the programmer to work on different abstraction levels.

I would like to work towards this vision, and the deep connection between programming (type theory) and category theory seems to be a promising tool. Particularly, because category theory makes extensive use of visualizations (in the form of diagrams), while also being a natural framework for repeated abstractions.

Furthermore, realizing this vision would involve writing an interpreter, and in order to eliminate errors, it should be written in a language with a strong typing system.

Therefore, I take this thesis as an opportunity to combine both aspects: By formalizing in Agda, I can implement an interpreter and formally verify its correctness, and then continue by exploring the connection between programming and category theory.

Structure

The following chapters are structured as follows: In chapter 2, Agda and its syntax are introduced as an example of working in a type theory. In chapter 3, some types which occur frequently are presented. In chapter 4, category theory is introduced and is developed far enough for the definition of a cartesian closed category. In chapter 5, the simply typed lambda calculus is introduced. This involves the definition of a typechecker, context weakening and substitution, as well as proofs about their behaviour. Furthermore, β-reduction is defined, and normalization of well typed terms is proven. Finally, in chapter 6, the interpretation of well typed terms in a CCC is given and proven to be sound with respect to β-reduction.

2 Formalization in Agda

2.1 About Agda

Agda is a dependently typed, functional programming language with a syntax similar to Haskell. It is being actively developed, with recent features including support for Cubical Type Theory, and a new, light-weight syntax for implicit arguments.

This thesis was written using Literal Agda source files, which combine LaTeX-markup and Agda code. While many parts of the code remain hidden in this final document, everything is still checked and formally verified to be correct by the Agda typechecker.

Some of the newer features are not yet available in the official Agda binaries. Instead, we use a self-compiled build of Agda from the master branch of its git repository [5]. The hash of the commit with which the code was tested is `fe6337817cd295f1b7a928b4865f1`.

During development, some code from standard libraries was used. These are the agda-prelude [4] and the demo library for CTT [2]. Additionally, a standalone implementation of Cubical Type Theory by Anders Mörtberg [16] and the accompanying proofs provided a reference for how basic properties of types could be proven in CTT. Most prominently, a proof of the Hedberg-Lemma, being indirectly used in many places, was taken from there.

2.2 Introduction to Agda

We now start with a general treatment of types, then switch over to the language of Agda for the introduction of concepts usually found in dependent type theories.

As a general reference, see Geuvers [12]. More in-depth information about Agda may be found in its online documentation [3].

Types and terms

The basic building blocks of a type theory are types and terms. A type is defined by specifying how terms of this type can be constructed. We write

$$t : T$$

if the term t has type T. There are two perspectives on how a type can be interpreted.

The first perspective on types is to view them as being similar to sets, and accordingly, terms of a type are called it's elements or inhabitants. But there are some differences to be aware of:

1. Sets are defined by the elements they *contain*, while types, by how inhabitants *can be constructed.* This means that we cannot simply reason about the entirety of terms "in" a type, for example by counting them.

2. Since terms are defined together with their type, they have no independent existence. It follows that a term can never be an element of multiple different types. Because of this, the question of whether $t : T$ holds is decidable (while $t \in T$ in general is not), and a statement of this kind can be checked by the typechecker.

The second perspective stems from the fact that propositions are also encodable in types. From this point of view, constructing an element $p : P$ is like constructing a verifiable proof p of the proposition P.

Universes

In dependent type theories, types themselves have a type. Such a "type of types" is called a universe and being written as $\mathcal{U}$. Because of having $\mathcal{U} : \mathcal{U}$ would lead to inconsistencies, there is usually a hierarchy of universes, denoted by universe levels ℓ, such that $\mathcal{U}_\ell : \mathcal{U}_{\ell+1}$.

Remark. In Agda, $\mathcal{U}$ is a function which takes a level parameter. Because of this, we write $\mathcal{U}\ \ell$ instead of $\mathcal{U}_\ell$. Still, for simplicity, we define the name of the first universe to be $\mathcal{U}_0$.

The types we usually work with do not contain other types inside of them, which means that they are small enough to live inside $\mathcal{U}_0$. Only categories are, for maximum generality, defined in a universe polymorphic way.

Defining simple types in Agda

In Agda, a type can be defined using the data keyword. It expects a name and a universe in which this type should live. In the following where-block, the constructors of this type need to be listed.

Example 2.1. A type with two constructors, remniscent of a set with two elements, can be defined as follows. The type is called Bool, it lives in $\mathcal{U}_0$ and has two constructors: true and false.

```
data Bool : 𝓤₀ where
  true  : Bool
  false : Bool
```

Continuing this way, we define a type with only one constructor, as well as a type without constructors at all.

Example 2.2.

i) The type ⊤ is called top. It has a single constructor tt.

```
data ⊤ : 𝓤₀ where
  tt : ⊤
```

ii) The type ⊥ is called bottom. It has no constructors.

```
data ⊥ : 𝓤₀ where
```

Following the interpretation of types as sets, these correspond, respectively, to the singleton set $\{*\}$ and the empty set $\varnothing$. If, instead, we view types as propositions, then ⊤ can be seen as truthhood, i.e., a trivially true proposition, whose proof can always be given by tt. The bottom type ⊥ then is falsehood, for which no proof can be given.

Statements

Having defined types and terms, they can now be used in statements. Statements simply assign a name to some term, but usually the type of this term has to be explictly given as well. Depending on the context they are used in, they may serve as simple renamings, definitions, or theorems and their proofs.

Example 2.3. We define 𝟚 to be an alternative name for Bool.

```
𝟚 : 𝓤₀
𝟚 :≡ Bool
```

We define el1 as an alternative name for true.

```
el₁ : 𝟚
el₁ :≡ true
```

Here, the types 𝟚 and Bool are definitionaly equal: the typechecker does not differentiate between these expressions. Speaking on a meta-theoretic level about Agda, we say 𝟚 ≡ Bool. This definitional equality is not the same as the (path-) equality, written $a = b$, which will be introduced later.

Functions

Given two types A and B, the type of functions between them is written as $A \to B$. A function term can be either constructed by a lambda expression, or directly as part of a statement.

Example 2.4. The identity function for Bool can be defined in the following, definitionally equal ways.

```
idB₁ : Bool → Bool
idB₁ :≡ λ b → b
```

```
idB₂ : Bool → Bool
idB₂ b :≡ b
```

A function is applied to arguments by writing them after each other.

```
true₂ : Bool
true₂ :≡ idB₁ true
```

Remark. Function application always takes precedence over other operations (except the evaluation of parentheses).

A function can be defined by pattern matching on the constructors of the argument type.

Example 2.5. Boolean negation is defined by pattern matching:

```
negate : Bool → Bool
negate true  :≡ false
negate false :≡ true
```

Functions with multiple arguments are usually defined as higher order functions, that is, as functions which return functions.

Example 2.6. The boolean conjunction is a function of type $\mathsf{Bool} \to (\mathsf{Bool} \to \mathsf{Bool})$. That is, a function taking a boolean and returning a function which takes another boolean, and returns the result. The function arrow associates to the right, consequently, the parentheses can be omitted.

```
and : Bool → Bool → Bool
and true  true  :≡ true
and true  false :≡ false
and false true  :≡ false
and false false :≡ false
```

Remark. Names can be turned into infix operators by writing an underscore where arguments are supposed to be placed, for example we define:

```
_∧_ : Bool → Bool → Bool
a ∧ b :≡ and a b
```

Furthermore, names can contain every possible mix of characters: different tokens are only distinguished by the whitespace between them. Accordingly, $a{\wedge}b$ is a name, while $a \wedge b$ is the application of the function $\wedge$ to the terms a and b. We often choose names such as $a{=}b$ or $i{<}n$ for terms which prove such statements.

The logical interpretation of a function $P \to Q$ is that of an implication: Being able to construct such a function means that a proof of P can be turned into a proof of Q. A proposition P is false if $P \to \bot$ can be proven, since this means that a proof of P would give us a proof of $\bot$, of which we know that it cannot exist.

Data types with arguments

When defining a type, constructors may take arguments. This effectively turns them into functions, and the syntax is the same.

Example 2.7. The type ℕ of natural numbers is defined as an inductive data type with two constructors: A natural number is either zero, or it is the successor of another natural number.

```
data ℕ : 𝒰₀ where
  zero : ℕ
  suc  : ℕ → ℕ
```

The meaning of *inductive* here is that a constructor recursively takes arguments of the type which it constructs.

Constructors are applied exactly like functions:

Example 2.8. The number 4 can be encoded as follows:

```
four : ℕ
four :≡ suc (suc (suc (suc zero)))
```

Pattern matching on constructors allows us to bring their arguments into scope by giving them a name.

Example 2.9. The operation of addition on ℕ can be defined by recursion.

```
_+ℕ_ : ℕ → ℕ → ℕ
zero    +ℕ b :≡ b
(suc a) +ℕ b :≡ suc (a +ℕ b)
```

Remark. By default, Agda allows only total functions. In order to enforce this, it has a termination checker which verifies that at least one argument of a recursive function call gets smaller in every iteration. Here, this is the case for the first argument, since a is smaller than suc a.

Functions with type parameters

Functions can take type parameters.

Example 2.10. The identity function can be defined for all types by letting it take a type parameter.

```
idf₁ : (A : 𝒰₀) → A → A
idf₁ A a :≡ a
```

Example 2.11. This function can be made universe polymorphic by requiring an additional level parameter. Here we pattern match with underscores, indicating that these arguments are not used in the function body.

```
idf₂ : (ℓ : ULevel) → (A : 𝒰 ℓ) → A → A
idf₂ _ _ a :≡ a
```

Remark. Agda provides a way to make arguments implicit by enclosing them with curly braces. Then, when calling such a function, these arguments do not have to be given, instead, Agda tries to infer their values from the context.

```
idf : {ℓ : ULevel} → {A : 𝒰 ℓ} → A → A
idf a :≡ a
```

Sometimes it is still necessary to give such arguments, or to pattern match against them. In both cases this can be done by using curly braces.

Global implicits

Agda has a new syntax which allows us to declare global implicit variables. They define variable names which can be used in function definitions as if they were implicit variables. This feature currently does not work with data types, where implicit arguments still have to be named individually.

Example 2.12. In order to declare ℓ and ℓ' as always being universe levels, we write:

```
variable
  {ℓ ℓ′} : ULevel
```

Data types with type parameters

Data types can take level and type parameters as well. These are stated directly after the name.

Example 2.13. Using this, we can define the product and the sum type.

(i) For two types A and B, we define the product $A \times B$ as the type which can be constructed by providing an element of A and an element of B. Because A and B can live in different universes $\mathcal{U}_\ell$ and $\mathcal{U}_{\ell'}$, the resulting type has to live in the one which is larger, namely $\mathcal{U}_{\mathsf{lmax}\,\ell\,\ell'}$.

```
data _×_ {ℓ ℓ′} (A : 𝒰 ℓ) (B : 𝒰 ℓ′) : 𝒰 (lmax ℓ ℓ′) where
  _,_ : A → B → A × B
```

(ii) For two types A and B, we define the sum $A + B$ as the type which can be constructed by either providing an element of A, or an element of B. The same note about universe levels applies.

```
data _+_ {ℓ ℓ′} (A : 𝒰 ℓ) (B : 𝒰 ℓ′) : 𝒰 (lmax ℓ ℓ′) where
  left  : A → A + B
  right : B → A + B
```

The corresponding notions in set theory are the cartesian product and the disjoint union of sets.

Viewed as an operation on propositions P and Q, the logical interpretation of $P \times Q$ is $P \wedge Q$, and of $P + Q$ it is $P \vee Q$. That said, the behaviour of $P + Q$ is slightly different from it's logical counterpart. An element $p : P + Q$ contains additional information about which proposition out of these two was proven [27].

Record types

Data types with only a single constructor are effectively tuples, containing (multiple) values. They can be defined more conveniently using record syntax. It differs from the data syntax in that the values, called fields, are given explicit names. These names define projection functions which can be used to access the respective values.

Example 2.14. The product type can be defined as a record. As before, the constructor is called _,_ and has the type $A \to B \to A \times B$. Additionally, the projection functions fst : A × B → A and snd : A × B → B are defined.

```
record _×_ {ℓ ℓ′} (A : 𝒰 ℓ) (B : 𝒰 ℓ′) : 𝒰 (lmax ℓ ℓ′) where
  constructor _,_
  field
    fst  : A
    snd : B
```

Terms of a record type can be constructed using a dedicated copattern syntax. For this, the value of every field has to specified, in a way similar to pattern matching.

Example 2.15. The pair of natural numbers $(0, 1)$ can be defined as follows.

```
pair : ℕ × ℕ
fst  pair :≡ zero
snd pair :≡ suc zero
```

Dependent types

A dependent type is a function which returns a type. It is also called a type family [18].

Example 2.16. T_1 is a type family, depending on an argument of type Bool.

```
T₁ : Bool → 𝒰₀
T₁ true  :≡ ⊤
T₁ false :≡ ℕ
```

Using type families, we can define functions whose resulting type depends on the arguments given.

Example 2.17. f_1 is a dependent function which takes a boolean argument and returns a term of T if the argument was true, and a natural number if it was false.

```
f₁ : (a : Bool) → T₁ a
f₁ true  :≡ tt
f₁ false :≡ four
```

Remark. The level and type polymorphic functions introduced before also represent special cases of dependent functions.

Dependent product

This operation of creating a function type out of a type family can be extracted into a new type: the dependent product.

Example 2.18. Given a type family B of type $A \to \mathcal{U}_{\ell'}$, the **dependent product** is the type of functions which for every $a : A$ return a term of type $B\,a$. The universe levels refer to the global implicits defined before.

```
Π : {A : 𝓤 ℓ} → (B : A → 𝓤 ℓ′) → 𝓤 (lmax ℓ ℓ′)
Π {A :≡ A} B :≡ (a : A) → B a
```

Usually, this type is written as follows:

$$\prod_{a:A} B\,a$$

The logical interpretation of the dependent product is that of the universal quantifier. A function of type $\prod_{x:X} P\ x$ has to give a proof of $P\ x$ for every possible $x : X$. This means that dependent products express the notion of universal quantification, $\forall (x \in X).\, P(x)$.

In Agda, we can write $\Pi\, B$ or $\Pi\, (\lambda\, a \to B\, a)$ or $\Pi\, (\lambda\, (a : A) \to B\, a)$, but usually we skip the product sign, and write it as the dependent function type $(a : A) \to B\, a$. Agda also allows an optional $\forall$ sign: $f_2 : \forall (a : A) \to B\, a$.

Dependent sum

The dependent sum is defined as a pair. Accordingly, we re-use the terminology from above.

Example 2.19. The **dependent sum** of a type family $B : A \to \mathcal{U}_{\ell'}$ is defined as a pair, where the type of the second element depends on the value of the first.

```
record Σ {ℓ ℓ′} {A : 𝒰 ℓ} (B : A → 𝒰 ℓ′) : 𝒰 (lmax ℓ ℓ′) where
  constructor _,_
  field
    fst : A
    snd : B fst
```

This type is usually written as follows:

$$\sum_{a:A} B\,a$$

In order to be able to construct a term of type $\sum_{x:X} P x$, we have to find some $x : X$, for which $P\,x$ is provable. Dually to the dependent product type, the logical interpretation of the dependent sum type is that of the existential quantifier $\exists (x \in X).\, P(x)$.

In Agda, we write this type as $\Sigma\, B$ or $\Sigma\, (\lambda\, a \to B\, a)$ or $\Sigma\, (\lambda (a : A) \to B\, a)$.

Equality

In dependent type theories, types can capture the notion of equality of elements. It is expressible by the following type family:

$$\_{=}\_ : \forall\{\ell\} \to \{A : \mathcal{U}_\ell\} \to A \to A \to \mathcal{U}_\ell$$

For a type $A : \mathcal{U}_\ell$ and elements $a\, b : A$, equality is therefore proven by constructing an element of $a = b$.

Depending on the specific type theory in use, the implementation of this type family varies. In Cubical Type Theory it is modeled by paths, as described in Cohen et al. [8]. Here, we only show the most basic principles of CTT, focusing more on the practical aspects of writing equality proofs.

On a topological space X, a path p is defined as a continuous function $p : [0,1] \to X$. Analoguously, in CTT, there is a type I with formal elements i0 and i1. Equalities on a type A are treated similar to functions I → A.

For example, by reflexivity, the equality $a = a$ must always hold. This is formalized by a constant path.

Example 2.20. The constant path is called refl. Paths are using the same syntax as functions.

```
refl : ∀{ℓ} → {A : 𝒰 ℓ} → {a : A} → a = a
refl {ℓ} {A} {a} :≡ λ i → a
```

Remark. Even though paths use the same syntax as functions, their behaviour is not the same. For example, we cannot pattern match on **i** and write different implementations for i0 and i1.

But there are operations on I which can be used to construct new paths. For example, we can write $\sim \mathbf{i}$. In the topological space analogy, this corresponds to $1 - \mathbf{i}$, but here, its effective meaning is that of invertion, mapping $\mathsf{i0}$ to $\mathsf{i1}$ and vice versa. Using this, we can express the symmetry of equality.

Example 2.21. The operation of inverting a path is called sym.

$$\begin{aligned}&\mathsf{sym} : \forall\{\ell\} \to \{A : \mathcal{U}\ \ell\} \to \{a\ b : A\} \to a = b \to b = a\\ &\mathsf{sym}\ p\ \mathbf{i} :\equiv p\ (\sim \mathbf{i})\end{aligned}$$

Using further cubical primitives, the composition of paths, corresponding to transitivity can be formalized:

Example 2.22. The operation of composing paths is called trans. It has the following type:

$$\mathsf{trans} : \forall\{\ell\} \to \{A : \mathcal{U}\ \ell\} \to \{a\ b\ c : A\} \to a = b \to b = c \to a = c$$

Notation. The composition of two paths $p : a = b$ and $q : b = c$ is usually denoted by $p \bullet q$. For this we write:

$$\_\bullet\_ :\equiv \mathsf{trans}$$

Another common way to modify an equality is to map a function over it.

Example 2.23. If $a = b$, then it is valid to apply a function f to both sides. This operation is called cong.

$$\begin{aligned}\mathsf{cong} : \forall\{\ell\ \ell'\} &\to \{A : \mathcal{U}\ \ell\} \to \{B : \mathcal{U}\ \ell'\} \to \{a\ b : A\}\\ &\to (f : A \to B)\\ &\to a = b \to f\ a = f\ b\\ \mathsf{cong}\ f\ p\ \mathbf{i} &:\equiv f\ (p\ \mathbf{i})\end{aligned}$$

All of the operations introduced so far (refl, sym, trans and cong) can be expressed in many dependent type theories, regardless of the specific implementation of equality. Thus, when using these, proofs can be written in an implementation independent way. Nevertheless, sometimes it is very useful to drop down to the explicit path notation, for example, when mapping a binary function over two paths simultaneously.

The next operation, functional extensionality, cannot be proven in standard ITT or HoTT. There, it can only be assumed as an axiom, i.e., as a function without implementation. In CTT, the proof is straightforward:

Example 2.24. Functional extensionality means that the equality of two functions f and g can be derived from the fact that they return the same result for every input.

```
funExt : ∀{ℓ ℓ′} → {A : 𝒰 ℓ} → {B : 𝒰 ℓ′}
                 → {f g : A → B}
                 → (∀(a : A) → f a = g a)
                 → f = g
funExt p i a :≡ p a i
```

Proofs

Now we can state theorems and proof them. For example, the associativity of the addition of natural numbers.

Example 2.25. Associativity is proven by the following function:

```
assoc : (a b c : ℕ) → (a +ℕ b) +ℕ c = a +ℕ (b +ℕ c)
assoc zero b c      :≡ refl
assoc (suc a′) b c :≡ cong suc (assoc a′ b c)
```

The proof can be explained as follows (we write $+$ instead of $\mathbb{N}+$).
We consider the cases $a \equiv 0$ and $a \equiv (\mathsf{suc}\, a')$ separately:

- For $a \equiv 0$, the goal reduces to $(0 + b) + c = 0 + (b + c)$.

  By the definition of $\_{+}\mathbb{N}\_$, $0 + b$ is simply b.

  Analoguously, $0 + (b + c)$ reduces to $b + c$.

  Therefore, the goal is $b + c = b + c$.

  We conclude with refl.

- For $a \equiv \mathsf{suc}\, a'$, the goal is $((\mathsf{suc}\, a') + b) + c = (\mathsf{suc}\, a') + (b + c)$.

  After evaluating the definition of $+\mathbb{N}$ two times on the left side and one time on the right side, the goal reduces to:

  $$\mathsf{suc}\, ((a' + b) + c) = \mathsf{suc}\, (a' + (b + c))$$

  By calling $\mathsf{assoc}\, a'\, b\, c$, we get a proof of:

  $$(a' + b) + c = a' + (b + c)$$

  We use cong suc in order to apply suc to both sides. This finishes the proof.

Longer proofs

For a slightly more complex example, we introduce the definition of the ordering relation $\_{<}\_$ on $\mathbb{N}$. For $n\, m : \mathbb{N}$, a proof of $n < m$ is given by the following type:

$$\sum_{k:\mathbb{N}} (m = \mathsf{suc}\, k + n)$$

Definition 2.26. In Agda, the ordering relation _<_ on ℕ is defined by the following record.

```
record _<_ (n m : ℕ) : 𝒰₀ where
  constructor diff
  field
    diff–k : ℕ
    diff–p : m = suc diff–k +ℕ n
```

◁

Remark. Later, we will need use some of its properties, including the fact that the ordering still holds after taking the successor or predecessor of both sides.

```
suc–monotone  : {k l : ℕ} → k < l → suc k < suc l
pred–monotone : {k l : ℕ} → suc k < suc l → k < l
```

Another property is antireflexivity, which can be proven using an operation called *substitution*.

```
less–antirefl : {n : ℕ} → n < n → ⊥
```

For an example of a longer proof, we now show the antisymmetry of _<_.

Example 2.27. The ordering relation _<_ is antisymmetric.

```
less–antisym : {n m : ℕ} → n < m → m < n → ⊥
less–antisym {n} {m} (diff k kp) (diff l lp) :≡
  let
    proof : n = suc (suc (l +ℕ k) +ℕ n)
    proof :≡ n                           ≡⟨ lp ⟩
             suc (l +ℕ m)                ≡⟨ cong (λ ξ → suc (l +ℕ ξ)) kp ⟩
             suc (l +ℕ (suc (k +ℕ n)))   ≡⟨ cong suc (add–suc–r l (k +ℕ n)) ⟩
             suc (suc (l +ℕ (k +ℕ n)))   ≡⟨ cong (suc ∘ suc) (sym (assoc l k n)) ⟩
             suc (suc ((l +ℕ k) +ℕ n))   ∎

    n<n : n < n
    n<n :≡ diff (suc (l +ℕ k)) proof

  in less–antirefl n<n
```

This proof uses a let … in clause to introduce two local bindings called proof and n<n. Then less–antirefl is called to get a proof of ⊥.

In the definition of proof, the operators _≡⟨_⟩ and ∎ provide a readable syntax for chaining paths together. The terms on the left side represent the intermediate steps of the derivation, just like it would be written manually. Internally, they are are discarded after typechecking, and the paths on the right side are composed using _•_.

As seen here, even small proofs get rather long very fast. Therefore, we will hide them most of the time, explaining the idea behind statements instead.

Contradictions

Using functions like less−antisym, we can, if given correct arguments, show that they lead to a contradiction. From such a contradiction, anything can be derived [27].

In Agda, when there are no valid constructors for an argument, empty parentheses can be used instead of a name. Then no function body has to be written.

```
⊥−elim : {A : 𝓤 ℓ} → ⊥ → A
⊥−elim ()
```

Comparing elements

Generally, for two elements a and b of a type A, the question of whether they are equal is not decidable. But sometimes it is necessary to require such a property, for example when defining the typechecker.

In order to formalize this, we first define the concept of decidability.

Definition 2.28. A type, viewed as a proposition, is called **decidable** if either a proof or a refutation can be given.

```
data isDec {ℓ} (A : 𝓤 ℓ) : 𝓤 ℓ where
  yes : A         → isDec A
  no  : (A → ⊥) → isDec A
```

◁

Now we can define what it means for a type to have comparable elements:

Definition 2.29. A type is called **discrete** if for every pair of elements, equality is decidable.

```
isDiscrete : (A : 𝓤 ℓ) → 𝓤 ℓ
isDiscrete A :≡ (x y : A) → isDec (x = y)
```

◁

3 Basic constructs

In this chapter we define some objects which will serve as basic building blocks later on.

3.1 Finite type

We often need a finite set of indices. For a given size n, such a type can be modelled by the sum of natural numbers i smaller than n, that is:

$$\sum_{i:\mathbb{N}} i < n$$

In Agda, in order to improve type inference and error messages, we explicitly define this type as a record, instead of reusing Σ.

Definition 3.1. Given a natural number n, we define a type Fin n, which has exactly n elements, as the type containing natural numbers smaller than n. Sometimes we refer to the elements as indices.

```
record Fin (n : ℕ) : 𝓤₀ where
  constructor _⌈_
  field
    º : ℕ
    ºless : º < n
```

$\triangleleft$

Elements of Fin n can be constructed by giving a natural number and a proof that it is smaller than n. For construction and pattern matching, the infix constructor _⌈_ is used.

Example 3.2. Common indices are:

(i) In every finite type with at least one element, there is an element fzero.

```
fzero : ∀{n} → Fin (suc n)
fzero :≡ zero ⌈ 0<suc
```

(ii) Given an index, we can construct its successor. It lives in the next greater finite type.

```
fsuc : ∀{n} → Fin n → Fin (suc n)
fsuc (k ⌈ k<n) :≡ (suc k) ⌈ (suc–monotone k<n)
```

(iii) Combining these functions, the element fone of finite types with at least two elements is defined as follows.

```
fone : ∀{n} → Fin (suc (suc n))
fone :≡ fsuc fzero
```

3.2 Finite lists

Definition 3.3. Given a type A, a **finite list over** A **of length** n is a function Fin $n \to A$, mapping indices to elements.

```
FList : 𝒰 ℓ → ℕ → 𝒰 ℓ
FList A n :≡ Fin n → A
```

◁

Remark. To construct a list Γ means to construct a function which, given an index i, returns the i-th element of this list. We access it by writing $\Gamma\, i$.

Example 3.4. We look at some examples for constructing lists.

(i) Empty list over A. We are given an index of type Fin 0 and have to return the element at this position. But being given such an index means we are given a natural number $k : \mathbb{N}$ and a proof that $k < 0$. This is a contradiction, so we can conclude by applying ⊥–elim.

```
[] : FList A 0
[] (k ⌈ k<0) :≡ ⊥–elim (lessZero–⊥ k<0)
```

(ii) Prepending x to Γ. The resulting list has x at index 0 (first case) and $\Gamma\, i$ at index $i+1$ (second case). In order to have an index of type Fin n which is accepted by Γ, we turn the proof of suc $i <$ suc n into a proof of $i < n$ by applying pred–monotone.

```
_,,_ : A → FList A n → FList A (suc n)
(x ,, Γ) (zero ⌈ _)      :≡ x
(x ,, Γ) (suc i ⌈ si<sn) :≡ Γ (i ⌈ pred–monotone si<sn)
```

(iii) Inserting x at index j. In order to compute the element at position i, we compare the natural number part of both indices. If i is smaller than j, which means that we are trying to access an element before the insertion point, we simply return the i-th element of the original list. If we are exactly at the point of insertion, we return the new element x. Else, if we are already past the point of insertion, we first decrement our given index by 1 (fpred), in order to access the correct element in the original list.

```
insertL : (Γ : FList A n) → Fin (suc n) → A → FList A (suc n)
insertL Γ j x i with compare (º i) (º j)
... | less i<j    :≡ Γ (fsmaller i j i<j)
... | equal i=j   :≡ x
... | greater i>j :≡ Γ (fpred i (mkNotZero i>j))
```

Notation. We denote the insertion of an element x at position j into a list Γ by:

```
_↓_ : Fin (suc n) → A → FList A n → FList A (suc n)
(j ↓ x) Γ :≡ insertL Γ j x
```

Remark. Lists may be defined in different ways. In addition to the definition shown above, we could also define a list inductively with two constructors:

- [] (the empty list)
- $x, , \Gamma$ (an element $x : A$ prepended to some list $\Gamma : \mathit{List}\, A$)

Depending on which definition is chosen, different list operations become easier to implement. With the former definition, accessing the i-th element of a List Γ for some index i : Fin n is simply $\Gamma\, i$. The latter definition makes prepending elements and writing functions which recurse on the head x and tail Γ easier.

Since we often need to access and insert elements in the middle of the list, we choose the former definition. But this comes with a cost; prepending an element to a list and then taking the tail is not definitionally equal to the original list:

$$\Gamma \not\equiv \mathsf{tail}\,(x\,,\,,\,\Gamma)$$

Instead, we often have to explicitly use the following equality:

```
tail= : (x : A) → (Γ : FList A n) → Γ = tail (x , , Γ)
```

3.3 Error handling

In functional programming languages, the sum type is useful for handling errors and exceptions. This functionality is implemented as part of a monad interface [28], but for the sake of brevity we specialize the following definitions to our use case.

For a type Err containing error information, the function type $A \to \mathit{Err} + B$ models a function which, given an element of A, either fails with an error of type Err, or succeeds with an element of B.

Given such a result, we can feed it into another possibly failing function by inspecting whether it is a success or a failure:

```
_≫=_ : ∀{ℓ} → {A B Err : 𝒰 ℓ}
              → (Err + A) → (A → Err + B) → Err + B
_≫=_ (left e)  f :≡ left e
_≫=_ (right a) f :≡ f a
```

We can also ignore the value of a successful result and only propagate errors:

```
_≫_ : ∀{ℓ} → {A B Err : 𝓤 ℓ}
          → (Err + A) → (Err + B) → (Err + B)
_≫_ (left e)    b :≡ left e
_≫_ (right _) b :≡ b
```

Furthermore, given two such functions, they can be chained together:

```
_>=>_ : ∀{ℓ} → {A B C Err : 𝓤 ℓ}
           → (A → Err + B) → (B → Err + C) → (A → Err + C)
_>=>_ f g a :≡ f a ≫= g
```

Our use-case of these potentially failing functions is the implementation of the type-checker. But having implemented it, we will also need to prove properties about its behaviour. In order to be able to do this, we introduce a type called FIR ("function is right"). An element of FIR a f proofs that the function f succeeds when given the argument a.

Definition 3.5. The type FIR is implemented by the following record:

```
record FIR {A B X : 𝓤₀} (a : A) (f : A → X + B) : 𝓤₀ where
  constructor _,fir,_
  field
    fir : B
    firProof : f a = right fir
```

◁

It is useful in the following way: We can define a function dosplit♦, which, given a proof that the composition f >=> g succeeds, returns seperate proofs for FIR a f and FIR b g. Similarly, we define a function eval♦ for joining both proofs back together.

4 Category theory

4.1 What is a category?

When studying different mathematical objects, a common pattern on what such theories are made of emerges: A mathematical structure is being accompanied by a notion of morphisms.

Examples may be found in different fields: in Algebra, where groups, rings, fields are studied, each structure comes with the definition of an appropriate, structure preserving homomorphism. In Linear Algebra, the morphisms between vector spaces are called linear maps. In Topology, topological spaces have continuous functions as morphisms between them, and in Analysis, there are smooth functions between smooth manifolds.

These morphisms, even though very different in their detailed definitions, have something in common: they all behave like functions - in so far that they have the following properties:

1. Composition: Morphisms with matching domain and codomain may be composed.
2. Identity: There is a morphism which behaves like the identity function.

In category theory, we study the case of having objects of a certain kind and morphisms behaving like functions between them. In order to do this, we consider all those objects and morphisms between them as a single structure, and call such a structure a category.

This means that, for example, there is the category **Grp** of groups and group homomorphisms. Similarly there are the categories **Ring**, **Fld**, **Top** and **Diff** [17]. And, as the archetypal category, there is **Set**, the category of sets and functions between them.

For introductory texts on category theory, see e.g. Awodey [6] or Smith [26]. The definitions in this chapter are based on the definitions found there.

Definition 4.1. A **category** is given by:

1. A type of objects Obj, and for every two objects $A\ B : \mathsf{Obj}$, a type of morphisms $\mathsf{Hom}\ A\ B$.
2. An identity morphism id, and a composition operation $\_\diamond\_$.
3. Proofs that the identity morphism is a left and right identity (unit) and that composition is associative.

We formalize this as a record:

```
record Category (ℓ ℓ′ : ULevel) : 𝓤 (lsucc (lmax ℓ ℓ′)) where
  field
    Obj    : 𝓤 ℓ
    Hom    : Obj → Obj → 𝓤 ℓ′

    id     : ∀{A} → Hom A A
    _⋄_    : ∀{A B C} → Hom A B → Hom B C → Hom A C

    unit-l : ∀{A B} → (f : Hom A B) → id ⋄ f = f
    unit-r : ∀{A B} → (f : Hom A B) → f ⋄ id = f
    asc    : ∀{A B C D}
             → (f : Hom A B) → (g : Hom B C) → (h : Hom C D)
             → (f ⋄ g) ⋄ h = f ⋄ (g ⋄ h)
```

◁

Remark. Usually, the composition operation is defined to compose backwards, like function composition does. In order to be more consistent with diagrams, we choose forward composition and denote it by _⋄_, instead of _∘_.

Notation. The morphisms between objects are also called arrows. We write this type as follows.

```
_⇁_ : Obj → Obj → 𝓤 ℓ′
A ⇁ B :≡ Hom A B
```

Example 4.2. For every universe level ℓ, the types and functions between them form a category.

```
open Category
Type : ∀ ℓ → Category (lsuc ℓ) ℓ
Obj     (Type ℓ) :≡ 𝓤 ℓ
Hom     (Type ℓ) :≡ λ A B → (A → B)
id      (Type ℓ) :≡ idf
(_⋄_)   (Type ℓ) :≡ λ f g → g ∘ f
unit-l  (Type ℓ) :≡ λ _ → refl
unit-r  (Type ℓ) :≡ λ _ → refl
asc     (Type ℓ) :≡ λ _ _ _ → refl
```

Remark (Diagrams). Often it is helpful to visualize configurations of arrows by drawing diagrams:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "h"] & B \arrow[d, "g"] \\ C \arrow[r, "i"] & D \end{tikzcd}$$

Such a diagram is said to **commute**, if, whereever possible, different paths from one object to another are equal. In this case, $f \diamond g = h \diamond i$ must hold.

4.2 Universal properties

In a category, the objects do not possess any internal structure. Still, different objects may be characterized by considering what arrows go into or come out of them.

By requiring the existence of certain unique arrows, the universal properties of different kinds of objects are formulated.

For example, the universal property of being a product object captures exactly the usual notions of products (e.g. products of groups or products of vector spaces).

We will consider three kinds of objects: terminal objects, products and exponentials.

Terminal object

Definition 4.3. An object X is **terminal** if, for every object A, there is a unique arrow $A \rightharpoondown X$.

```
isTerminal : Obj → 𝒰 (lmax i j)
isTerminal X :≡ ∀ A → Σ (λ (h : A ⇁ X) → Π (λ (k : A ⇁ X) → h = k))
```

◁

Remark. In this definition we implicitly work inside some category $\mathcal{C}$. The “*Obj*” and the arrow “⇁” refer to objects and arrows of this category. Still, when using the above function, the category has to be explicitly given as an argument.

Definition 4.4. A category $\mathcal{C}$ **has a terminal object** if there exists an object which is terminal. This object is called **1**. The unique arrow is called !, and the proof of uniqueness is called !–uprop.

```
record hasTerminal {i j} (𝒞 : Category i j) : 𝒰 (lsuc (lmax i j)) where
  open Category 𝒞
  field
    𝟏           : Obj
    𝟏isTerminal : isTerminal 𝒞 𝟏

  ! : {X : Obj} → X ⇁ 𝟏
  ! {X}              :≡ fst (𝟏isTerminal X)

  !–uprop : ∀{X} → {f : X ⇁ 𝟏} → ! = f
  !–uprop {X} {f} :≡ snd (𝟏isTerminal X) f
```

◁

Example 4.5. The category $\mathbf{Type}_{\ell}$ has a terminal object. It is ⊤, lifted to the level ℓ by Lift. The unique function to ⊤ is the one which ignores its argument and simply returns tt.

```
Type–hasTerminal                     : hasTerminal (Type ℓ)
1             (Type–hasTerminal) :≡ Lift ⊤
1isTerminal (Type–hasTerminal) :≡ λ A → !₀ , !₀–uprop
  where
    !₀ : {A : Obj} → (A ⇁ Lift ⊤)
    !₀ :≡ λ _ → (lift tt)

    createObjectPaths : {A : Obj} → (g : A → Lift ⊤) → (a : A) → g a = !₀ a
    createObjectPaths g a with (g a)
    ... | (lift tt) :≡ refl

    !₀–uprop : {A : Obj} → (g : A → Lift ⊤) → g = !₀
    !₀–uprop g :≡ funExt (createObjectPaths g)
```

Products

In order to define the product of two objects X and Y in a category $\mathcal{C}$, we consider, without explicitly constructing it, another category, where the objects are wedges to X and Y and the morphisms between them are arrows which make a certain diagram commute.

Definition 4.6. A **wedge** to X and Y is an object wObj together with a pair of arrows to X and Y.

$$\begin{tikzcd}
 & X \\
\mathsf{wObj} \arrow[ur, "w\pi_1"] \arrow[dr, "w\pi_2"'] & \\
 & Y
\end{tikzcd}$$

```
record Wedge (X Y : Obj) : 𝒰 (lmax i j) where
  constructor wedge
  field
    wObj : Obj
    wπ₁   : wObj ⇁ X
    wπ₂   : wObj ⇁ Y
```

◁

Definition 4.7. Given two wedges to X and Y, a **morphism of wedges** between them is a morphism f between their objects such that the following diagram commutes.

$$\begin{tikzcd}
 & X & \\
P \arrow[ur, "p_1"] \arrow[rr, "f"] \arrow[dr, "p_2"'] & & Q \arrow[ul, "q_1"'] \arrow[dl, "q_2"] \\
 & Y &
\end{tikzcd}$$

```
WedgeMorph : {X Y : Obj} → Wedge X Y → Wedge X Y → 𝓤 (j)
WedgeMorph (wedge P p₁ p₂) (wedge Q q₁ q₂)
  :≡ Σ (λ (f : P ⇀ Q) → (f ⋄ q₁ = p₁) |×| (f ⋄ q₂ = p₂))
```

◁

Remark. Here, we renamed the product type _ × _ to _|×|_, in order to use this name for the product object.

Definition 4.8. A wedge Z is called the **product of X with Y** if it is terminal in the category of wedges to X and Y.

That is, if for every wedge A there is a unique morphism $h : A \rightharpoonup Z$.

```
isProduct : {X Y : Obj} → Wedge X Y → 𝓤 (lmax i j)
isProduct {X} {Y} Z :≡ ∀ A → Σ (λ (h : WedgeMorph A Z)
                               → Π (λ (k : WedgeMorph A Z)
                               → fst h = fst k))
```

◁

Definition 4.9. A category $\mathcal{C}$ **has all products** if there is a binary operation _ × _ on objects, together with projection functions π_1 and π_2, such that for every pair of objects X and Y the wedge defined by $(X \times Y, \pi_1, \pi_2)$ is a product.

```
record hasProducts {ℓ ℓ′} (𝒞 : Category ℓ ℓ′) : 𝓤 (lsuc (lmax ℓ ℓ′)) where
  open Category 𝒞

  infixr 100 _×_
  field
    _×_        : Obj → Obj → Obj
    π₁         : ∀{A B} → A × B ⇀ A
    π₂         : ∀{A B} → A × B ⇀ B
    ×isProduct : ∀{A B} → isProduct 𝒞 (wedge (A × B) π₁ π₂)
```

For two arrows $f : A \rightharpoonup B$ and $g : A \rightharpoonup C$, we denote the unique arrow into $A \times B$ by $\langle f, g \rangle$.

```
⟨_,_⟩ : {A B C : Obj} → (A ⇀ B) → (A ⇀ C) → (A ⇀ B × C)
⟨_,_⟩ {A :≡ A} f g :≡ fst (fst (×isProduct (wedge A f g)))
```

The proof of its property of being a product is called ⟨, ⟩−prop.

```
⟨,⟩-prop : ∀{A B C : Obj} → (f : A ⇁ B) → (g : A ⇁ C)
         → (⟨ f , g ⟩ ⋄ π₁ = f) |×| (⟨ f , g ⟩ ⋄ π₂ = g)
⟨,⟩-prop {A} f g :≡ snd (fst (×isProduct (wedge A f g)))
```

And the proof of uniqueness is called ⟨,⟩−uprop.

```
⟨,⟩-uprop : ∀{A B C : Obj}
            → {f : A ⇁ B} → {g : A ⇁ C} → (h : A ⇁ B × C)
            → (h ⋄ π₁ = f) |×| (h ⋄ π₂ = g)
            → ⟨ f , g ⟩ = h
⟨,⟩-uprop {A} {f :≡ f} {g} h hprop
            :≡ snd (×isProduct (wedge A f g)) (h , hprop)
```

◁

Definition 4.10. For morphisms $f : A \rightharpoondown C$ and $g : B \rightharpoondown D$ we define the **morphism between the products** $A \times B$ **and** $C \times D$ by:

```
_××_ : ∀{A B C D} → (A ⇁ C) → (B ⇁ D) → (A × B ⇁ C × D)
_××_ f g :≡ ⟨ π₁ ⋄ f , π₂ ⋄ g ⟩
```

◁

Example 4.11. The category $\mathbf{Type}_\ell$ has products. They are given by the product type _|×|_ together with the projections fst and snd.

```
Type-hasProducts : hasProducts (Type ℓ)
_×_        (Type-hasProducts) :≡ _|×|_
π₁         (Type-hasProducts) :≡ fst
π₂         (Type-hasProducts) :≡ snd
×isProduct (Type-hasProducts) {X} {Y} (wedge A a₁ a₂) :≡
  let
    h : A → X |×| Y
    h a :≡ a₁ a , a₂ a

    H : WedgeMorph (Type ℓ) (wedge A a₁ a₂) (wedge (X |×| Y) fst snd)
    H :≡ h , (refl , refl)

    proof : ∀ K → fst H = fst K
    proof K i a :≡ (fst (snd K) (~ i) a , snd (snd K) (~ i) a)

  in H , proof
```

Exponentials

In the category $\mathbf{Type}_\ell$ we have the special case that the type of morphisms between two objects $X\ Y : \mathcal{U}_\ell$ is itself a type $X \to Y : \mathcal{U}_\ell$, and thus an object of $\mathbf{Type}_\ell$. The same happens in **Set**, where the functions from X to Y form a set, sometimes being denoted by Y^X.

Unlike product objects, such exponential objects are not as widespread. For example, in **Grp**, the group homomorphisms between two groups do not necessarily form a group themselves.

In order to define the property of an object being *like morphisms from X to Y*, we consider the category of evaluation structures between X and Y. (Again, without an explicit construction.)

Definition 4.12. An **evaluation structure between X and Y** is an object eObj together with an evaluation map eEv for it.

```
record Eval (X Y : Obj) : 𝓤 (lmax ℓ ℓ′) where
  constructor eval
  field
    eObj : Obj
    eEv  : eObj × X ⇀ Y
```

◁

Definition 4.13. A **morphism of evaluation structures between X and Y** is given by a morphism f between their objects which makes the following diagram commute.

$$\begin{tikzcd}
A \times X \arrow[d, "f \times\times \mathsf{id}"'] \arrow[dr, "a"] & \\
 & Y \\
B \times X \arrow[ur, "b"'] &
\end{tikzcd}$$

```
EvalMorphism : {X Y : Obj} → Eval X Y → Eval X Y → 𝓤 ℓ′
EvalMorphism (eval A a) (eval B b) :≡ Σ (λ (f : A ⇀ B) → (f ×× id) ⋄ b = a)
```

◁

Definition 4.14. An evaluation structure Z between X and Y is called the **exponential object of Y with X** if it is terminal in the category of evaluation structures.

That is, if for every evaluation structure A there exists a unique morphism from A to Z.

```
isExp : {X Y : Obj} → Eval X Y → 𝓤 (lmax ℓ ℓ′)
isExp Z :≡ ∀ A → Σ (λ (h : EvalMorphism A Z)
                → Π (λ (k : EvalMorphism A Z)
                → fst h = fst k))
```

◁

Definition 4.15. A category $\mathcal{C}$ **has exponentials** if there is a binary operation on objects _ˆ_, and for every pair of objects X and Y, an evaluation map for $Y \,\hat{}\, X$, such that the resulting evaluation structure is the exponential object of Y with X.

```
record hasExponentials {ℓ ℓ′} (𝒞 : Category ℓ ℓ′) (Products : hasProducts 𝒞)
                        : 𝒰 (lsuc (lmax ℓ ℓ′)) where
  open Category 𝒞
  open hasProducts Products

  field
    _ˆ_    : Obj → Obj → Obj
    ev     : ∀{X Y} → X ˆ Y × Y ⇁ X
    ˆisExp : ∀{X Y} → isExp 𝒞 Products (eval (X ˆ Y) ev)
```

For an arrow $f : A \times Y \rightharpoondown X$, the operation of getting the unique arrow $A \rightharpoondown X \,\hat{}\, Y$ is called currying.

```
  curry : {A X Y : Obj} → (A × Y ⇁ X) → (A ⇁ X ˆ Y)
  curry {A} f :≡ fst (fst (ˆisExp (eval A f)))
```

The proofs of being a morphism of evaluation structures and of uniqueness are called curry−prop and curry−uprop respectively.

```
  curry−prop : {A X Y : Obj}
             → (f : A × Y ⇁ X)
             → (curry f ×× id) ⋄ ev = f
  curry−prop {A} f :≡ snd (fst (ˆisExp (eval A f)))

  curry−uprop : {A X Y : Obj}
              → {f : A × Y ⇁ X} → (g : A ⇁ X ˆ Y)
              → (g ×× id) ⋄ ev = f
              → curry f = g
  curry−uprop {A} {X} {Y} {f} g p :≡ snd (ˆisExp (eval A f)) (g , p)
```

◁

Example 4.16. The category $\mathbf{Type}_\ell$ has exponential objects. For two types B and A, they are given by the function type $A \to B$.

We first define the evaluation function ev$_0$, which applies an argument to a function.

```
ev₀ : {A B : 𝒰 ℓ} → ((A → B) × A) → B
ev₀ (f , x) :≡ f x
```

Now we can prove that, indeed, all exponential objects exist. Currying is done by waiting for two arguments, and then combining them into a tuple. The curry−prop is trivially true. Uniqueness follows from the property of morphisms of evaluation structures.

```
Type-hasExponentials : hasExponentials (Type ℓ) (Type-hasProducts)
_^_    Type-hasExponentials :≡ λ B A → (A → B)
ev     Type-hasExponentials :≡ ev₀
^isExp Type-hasExponentials
       :≡ λ {(eval A f) → ((curry₀ f , curry-prop₀ f) , curry-uprop₀)}
  where
    curry₀ : ∀{A B C} → (A × B → C) → A → (B → C)
    curry₀ f :≡ λ a b → f (a , b)

    curry-prop₀ : ∀{A B C} → (f : A × B → C) → (curry₀ f ×× id) ◇ ev₀ = f
    curry-prop₀ f :≡ refl

    curry-uprop₀ : ∀{A B C} → {f : A × B → C}
                   → (k : EvalMorphism (Type ℓ) (Type-hasProducts)
                                          (eval A f) (eval (B → C) ev₀))
                   → (curry₀ f = fst k)
    curry-uprop₀ {f :≡ f} k i a b :≡ snd k (~ i) (a , b)
```

4.3 Cartesian closed categories

We have now explored exactly the kinds of objects which will be of further interest to us when we are going to provide a model for the lambda calculus.

There is a special term for referring to such categories:

Definition 4.17. A category $\mathcal{C}$ is called **cartesian closed** (short: it is a CCC) if it has a terminal object, all products and all exponentials.

```
record isCCC {i j : ULevel} (𝒞 : Category i j) : 𝒰 (lsucc (lmax i j)) where
  field
    Terminal     : hasTerminal 𝒞
    Products     : hasProducts 𝒞
    Exponentials : hasExponentials 𝒞 Products
```

◁

Combining the previous examples, we can show:

Example 4.18. The category $\mathbf{Type}_\ell$ is cartesian closed.

```
Type-isCCC : isCCC (Type ℓ)
Terminal     Type-isCCC :≡ Type-hasTerminal
Products     Type-isCCC :≡ Type-hasProducts
Exponentials Type-isCCC :≡ Type-hasExponentials
```

4.4 Finite products

Using our previously introduced concepts, we define a similar, very helpful object, the finite product. As the name implies, it may be constructed by repeatedly taking the (binary) product. But in order for this to be well-defined, the case of taking the product of zero objects also needs to be considered.

Definition 4.19. In a CCC, the **finite product of objects** is defined as a function which given a finite list of objects A, calculates their product $\prod A$ by recursion on the size of the list. The product of an empty list is the terminal object $\mathbf{1}$.

```
∏ : ∀{n} → (Fin n → Obj) → Obj
∏ {zero}   A :≡ 1
∏ {suc n} A :≡ ∏ (λ i → A (fsuc i)) × A fzero
```

◁

Similarly, by recursion on the size of the list, and by invoking the corresponding functions for binary products, we define finite projections and finite products of morphisms.

Definition 4.20. For a finite list of objects A, the **projection function of finite products** π_i, which projects the i-th element of the finite product $\prod A$, is defined as:

```
πᵢ : ∀{n} → {A : Fin n → Obj} → (i : Fin n) → ∏ A ⇁ A i
πᵢ {zero}   {A} (fin i (diff k p)) :≡ ⊥−elim (zNotS p)
πᵢ {suc n} {A} (fin zero p)        :≡ π₂ ⋄ O=⟦ cong A (finEqual zero) ⟧
πᵢ {suc n} {A} (fin (suc i) p)     :≡ π₁ ⋄ πᵢ (fin i (pred−monotone p))
                                          ⋄ O=⟦ cong A (finEqual (suc i)) ⟧
```

◁

Remark. Here, zNotS is a function which constructs a contradiction from a proof of zero $=$ suc n. The operator O=⟦ _ ⟧ takes an equality of objects $A = B$ as argument and returns an arrow $A \rightharpoondown B$. The function finEqual takes a natural number as input and returns a proof of equality for finite indices represented by this number.

Definition 4.21. For an object A, a finite list of objects B, and a finite list of morphisms $F_i : A \rightharpoondown B_i$, the **finite product of morphisms** $\langle\!\langle F \rangle\!\rangle$ of type $A \rightharpoondown \prod B$ is defined by:

```
⟪_⟫ : ∀{n} → {A : Obj} → {B : Fin n → Obj}
          → (F : (i : Fin n) → A ⇁ B i)
          → A ⇁ ∏ B
⟪_⟫ {zero}   F :≡ !
⟪_⟫ {suc n} F :≡ ⟨ ⟪ (λ i → F (fsuc i)) ⟫ , F fzero ⟩
```

◁

Definition 4.22. For a finite list of functions $F_i : A_i \rightharpoondown B_i$, the **morphism between finite products** $\bigtimes F$ of type $\prod A \rightharpoondown \prod B$ is defined by:

$$
\begin{aligned}
&\bigtimes : \forall\{n\} \to \{A\ B : \mathsf{Fin}\ n \to \mathsf{Obj}\} \to (F : \forall\ i \to A\ i \rightharpoondown B\ i) \\
&\qquad\qquad \to \prod\ A \rightharpoondown \prod\ B \\
&\bigtimes\ F :\equiv \langle\!\langle\ (\lambda\ i \to \pi_{\mathsf{i}}\ i \diamond F\ i)\ \rangle\!\rangle
\end{aligned}
$$

◁

5 Simply typed λ-calculus

When we restrict the type theory of Agda to only function types, together with lambda abstraction and application, we get the simply typed λ-calculus ($\lambda\!\to$). It is also called simple type theory (STT) and was first published by Church in 1940 [7].

The formalization in this chapter is based on the definitions found in Geuvers [12] and Pitts [20].

5.1 Parametrization

In Agda, we could define custom data types and their terms inside the type theory. This is not possible in $\lambda\!\to$, where the basic types and terms have to be chosen beforehand. Since this choice is arbitrary and does not affect the underlying theory, we parametrize $\lambda\!\to$ over every possible choice.

Definition 5.1. The simply typed λ-calculus is parametrized by:

- A type Gnd, whose elements will be the ground types. Since types have to be comparable, Gnd has to be discrete.
- A type Const, whose elements will be the constant terms.
- A function ctype mapping a constant to its type.

This parametrization is formalized by the following record:

```
record LambdaParam (ℓ ℓ′ : ULevel) : 𝒰 (lsuc (lmax ℓ ℓ′)) where
  constructor lambdaParam
  field
    Gnd        : 𝒰₀
    Gnd–isDisc : isDiscrete Gnd
    Const      : 𝒰₀
    ctype      : Const → Gnd
```

◁

The following sections all assume that such a parametrization has been given.

5.2 Types

The types of $\lambda\!\to$ can be either constructed by taking ground types or by forming a function type between two other types.

Definition 5.2. A type of λ→ is either an element of Gnd (ι), or a pair of types (_⇒_).

```
data Ty : 𝒰₀ where
  ι     : Gnd → Ty
  _⇒_ : Ty → Ty → Ty
```

◁

The typing of terms depends on what context they appear in. In order to describe a context, it is sufficient to state what variables are in scope and what their types are. Here we give the definition of a context. Its meaning will be explained in the next section.

Definition 5.3. A **context** is given by a finite list of types:

```
Ctx : ℕ → 𝒰₀
Ctx n :≡ Fin n → Ty
```

◁

5.3 Terms

Just like types, terms are defined as an inductive data type. Consequently, λ→ programs can be constructed directly in Agda by constructing an element of this type.

Definition 5.4. The terms of λ→ are defined as follows:

```
data Term : 𝒰₀ where
  cconst : Const → Term
  V      : ℕ → Term
  Λ      : Ty → Term → Term
  app    : Term → Term → Term
```

◁

Notation. The relation of a term t being well-typed and having the type τ in the context Γ is written as:

$$\Gamma \vdash t :: \tau$$

In order to distinguish this typing relation from Agda's own types, we use a double colon instead of a single one.

We now discuss the different constructors of terms together with their typing rules.

Constants

The constant terms and their types depend on the parametrization of λ→. A constant term can be constructed with an element of Const, its type is determined by ctype. As such, the type does not depend at all on the context in which the term appears. We write, for a context $\Gamma : \mathsf{Ctx}\, n$:

$$\frac{}{\Gamma \vdash \mathsf{cconst}\, c :: \iota\, (\mathsf{ctype}\, c)}$$

Remark. This presents a derivation rule, describing how the typing relation _ ⊢ _ :: _ should behave. From the hypothesis (above the line), the conclusion (below the line) can be derived. For simplicity, we do not include the conditition of elements having a certain type when it can be inferred from their usage. For example, here, c should be of type Const.

Variables

Variables are not represented by names, but by natural numbers, so called de Brujin indices. These are not arbitrary, but depend on the location where the variable was introduced. This way, we skip the notion of α-equivalence of terms, which else would be needed in order to group terms that use different variable names but are otherwise equal into equivalence classes. Using de Brujin indices, such an equivalence class collapses to a unique representation.

A context is a list of variables currently in scope, represented by their type. A variable term can be constructed with $\mathsf{V} : \mathbb{N} \to \mathsf{Term}$, by specifying the index in the context which we want to access.

A variable term can contain any natural number, but it is only well-typed if there actually is such a variable in the context. We write, for a context $\Gamma : \mathsf{Ctx}\, n$:

$$\frac{i < n}{\Gamma \vdash \mathsf{V}\, i :: \Gamma_i}$$

Abstraction

Lambda abstraction introduces a new variable into the context, which then can be used by the term inside. Outside of the lambda, this corresponds to a function taking such an argument.

Our version of $\lambda\!\to$ is “Church-style”. This means that, when creating a lambda abstraction, the type of the newly introduced variable has to be explicitly stated, as opposed to “Curry-style”, where it can be inferred by the typechecker instead. But this would mean additional complexity in the typechecker, which we choose to avoid, accepting the cost of slightly more verbose programs.

So, in order to construct such a lambda-abstraction-term, the constructor $\Lambda : \mathsf{Ty} \to \mathsf{Term} \to \mathsf{Term}$ has to be given the type of the new variable and the term for the function body. Here we use an uppercase Λ, because the lowercase version is already taken by Agda’s own lambda abstraction.

For a context $\Gamma : \mathsf{Ctx}\, n$, the typing rule is given by:

$$\frac{(\sigma,,\Gamma) \vdash t :: \tau}{\Gamma \vdash \Lambda_\sigma\, t :: (\sigma \Rightarrow \tau)}$$

Remark. The new variable is inserted at the beginning of the context. This means that the indices of all previously existing variables get incremented. As a result, the index by which a variable has to be accessed depends on the location where it is accessed from.

Application

The constructor for function application is $\mathsf{app} : \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term}$. It has to be given the term of the function and the term of the argument. Such an application is well-typed if the type of the argument matches the domain type of the function.

For a context $\Gamma : \mathsf{Ctx}\, n$, the typing rule is given by:

$$\frac{\Gamma \vdash t :: (\sigma \Rightarrow \tau) \qquad \Gamma \vdash s :: \sigma}{\Gamma \vdash \mathsf{app}\, t\, s :: \tau}$$

5.4 The typechecker

Based on the typing rules formulated above, we present a typechecking algorithm for $\lambda{\to}$. It utilizes the sum type for error handling, as described in section 3.3.

For this, we need to define a type which is going to contain the error information, i.e., why a term was incorrectly typed.

Definition 5.5. The following type errors may occur:

```
data TypeError : 𝒰₀ where
  ErrTypeMismatch   : Ty → Ty → TypeError
  ErrNoSuchVariable : ℕ → TypeError
  ErrIsNoFunction   : TypeError
```

◁

During typechecking, different conditions need to be asserted. This is done in auxilliary functions.

Definition 5.6. The auxilliary functions for typechecking are defined as follows:

(i) The function testTypeEq checks whether two given types are equal. Here, =stype= is used to compare types with each other. It is a proof of isDiscrete Ty, which itself is derived from the requirement of ground types to be discrete.

```
testTypeEq : Ty → Ty → TypeError + ⊤
testTypeEq σ τ with σ =stype= τ
... | yes _ :≡ right tt
... | no _  :≡ left (ErrTypeMismatch σ τ)
```

(ii) The function testFin checks whether a given natural number refers to a valid variable. If it is valid, the corresponding index for accessing the context is returned, or else, an error.

```
testFin : (n : ℕ) → (i : ℕ) → TypeError + Fin n
testFin n i with compare i n
... | less (i<n) :≡ right (fin i i<n)
... | equal _    :≡ left (ErrNoSuchVariable i)
... | greater _  :≡ left (ErrNoSuchVariable i)
```

(iii) The function testFunctionType checks whether a given type is a function type, and if it is, returns the domain and target types, or else, an error.

```
testFunctionType : Ty → TypeError + (Ty × Ty)
testFunctionType (ι _)     :≡ left (ErrIsNoFunction)
testFunctionType (σ ⇒ τ) :≡ right (σ , τ)
```

◁

Definition 5.7. The typechecker of $\lambda{\to}$ is defined by two mutually recursive functions. syn′ synthesizes the type of a term in a context. check′ checks whether a term has a given type in a context. The idea for such an architecture is taken from Dunfield and Krishnaswami [11].

```
syn′   : Ctx n → Term → TypeError + Ty
check′ : Ctx n → Term → Ty → TypeError + T
```

Since our λ-abstractions are "Church-style", types can actually be fully synthesized. This leads to a simple checking function. It only has to check whether the inferred type of a term is equal to the stated type.

```
check′ Γ t τ :≡ syn′ Γ t ≫= testTypeEq τ
```

The synthesizing is done as follows:

```
syn′ {n} Γ (Λ σ t)     :≡ syn′ (σ ,, Γ) t ≫= (λ τ → right (σ ⇒ τ))
syn′ {n} Γ (app t s)   :≡ syn′ Γ t
                            ≫= testFunctionType
                            ≫= λ {(σ , τ) → check′ Γ s σ ≫ right τ}
syn′ {n} Γ (V i)       :≡ testFin n i ≫= (right ∘ Γ)
syn′ {n} Γ (cconst c) :≡ right (ι (ctype c))
```

◁

We want to be able to use the FIR type for expressing the fact that type checking or synthesizing succeeds. In order to do this, we define alternative functions, taking a single tuple as argument.

```
syn : Ctx n × Term → TypeError + Ty
syn (Γ , t) :≡ syn′ Γ t
```

```
check : Ctx n × Term × Ty → TypeError + ⊤
check (Γ , t , A) :≡ check′ Γ t A
```

Notation. We denote successful typechecking of the term t with the type τ in the context Γ by $\Gamma \vdash t :: \tau$.

```
_⊢_::_ : (Γ : Ctx n) → Term → Ty → 𝓤₀
Γ ⊢ t :: τ :≡ FIR (Γ , t , τ) check
```

5.5 Typing proofs

We show that the typechecker behaves exactly as the typing rules stated above require. This means that, for every rule, we can prove implications up and down: from the hypothesis to the derivation and also, from a valid derivation back to the hypothesis.

Since such derivation rules describe implications, they can be formalized using functions:

Theorem 5.8. *The typechecker respects the typing rules given above.*

(i) The constant rule.

```
cconst⇓ : ∀{c τ} → {Γ : Ctx n} → ι (ctype c) = τ
                               → Γ ⊢ (cconst c) :: τ

cconst⇑ : ∀{c τ} → {Γ : Ctx n} → Γ ⊢ (cconst c) :: τ
                               → ι (ctype c) = τ
```

(ii) The variable rule. Here, in the case of V⇑*, the existence of an index j can be stipulated since we know that the natural number i accesses a valid variable. The property of $i < n$ is implicitly contained in i being used as an index into Γ.*

```
V⇓ : ∀{i τ} → {Γ : Ctx n} → Γ i = τ
                          → Γ ⊢ V (o̲ i) :: τ

V⇑ : ∀{i τ} → {Γ : Ctx n} → Γ ⊢ (V i) :: τ
                          → Σ (λ j → (Γ j = τ) × (o̲ j = i))
```

(iii) The lambda rule. Here it is useful to consider the cases of a lambda term having a ground type (Λ⇑ι) or a function type (Λ⇑⇒) individually.

```
Λ⇓    : ∀{t σ τ}    → {Γ : Ctx n} → (σ ,, Γ) ⊢ t :: τ
                                  → Γ ⊢ (Λ σ t) :: σ ⇒ τ

Λ⇑ι   : ∀{t σ c}    → {Γ : Ctx n} → Γ ⊢ (Λ σ t) :: (ι c)
                                  → ⊥
```

$$
\begin{aligned}
\Lambda{\Uparrow}{\Rightarrow} : \forall\{t\ \sigma\ \sigma_2\ \tau\} &\to \{\Gamma : \mathsf{Ctx}\ n\} \\
&\to \Gamma \vdash (\Lambda\ \sigma\ t) :: (\sigma_2 \Rightarrow \tau) \\
&\to (\sigma_2\ ,,\ \Gamma \vdash t :: \tau) \times (\sigma_2 = \sigma)
\end{aligned}
$$

$$
\begin{aligned}
\Lambda{\Uparrow} \quad : \forall\{t\ \sigma\ \psi\} \quad &\to \{\Gamma : \mathsf{Ctx}\ n\} \\
&\to (\Gamma \vdash \Lambda\ \sigma\ t :: \psi) \\
&\to \Sigma\ (\lambda\ \tau \to (\sigma\ ,,\ \Gamma \vdash t :: \tau) \times ((\sigma \Rightarrow \tau) = \psi))
\end{aligned}
$$

(iv) The application rule.

$$
\begin{aligned}
\mathsf{app}{\Downarrow} : \forall\{s\ t\ \sigma\ \tau\} &\to \{\Gamma : \mathsf{Ctx}\ n\} \\
&\to (\Gamma \vdash t :: \sigma \Rightarrow \tau) \to (\Gamma \vdash s :: \sigma) \\
&\to \Gamma \vdash \mathsf{app}\ t\ s :: \tau
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{app}{\Uparrow} : \forall\{s\ t\ \tau\} \quad &\to \{\Gamma : \mathsf{Ctx}\ n\} \\
&\to \Gamma \vdash \mathsf{app}\ t\ s :: \tau \\
&\to \Sigma\ (\lambda\ \sigma \to (\Gamma \vdash t :: \sigma \Rightarrow \tau) \times (\Gamma \vdash s :: \sigma))
\end{aligned}
$$

Proof. These statements are proven using the FIR type and the related dosplit♦ and eval♦ functions. □

The typing properties of $\lambda{\to}$ can now be used to prove further statements, such as the following.

Theorem 5.9. *If a term t is well-typed, then its type is uniquely determined.*

$$
\begin{aligned}
\mathsf{uniqueT} : \forall\{t\ \tau\ \upsilon\} \to \{\Gamma : \mathsf{Ctx}\ n\} &\to (T : \Gamma \vdash t :: \tau) \to (U : \Gamma \vdash t :: \upsilon) \\
&\to \tau = \upsilon
\end{aligned}
$$

5.6 Weakening

Given a term $\Gamma \vdash t :: \tau$, it can be modified to be valid in contexts which are weaker than Γ, that is, contexts which contain additional variables. Using list operations, such a weakened context is denoted by $(j \downarrow \sigma)\,\Gamma$, meaning the context obtained by inserting the type σ at position j into Γ.

Considering now the term t, we need to update it accordingly, because the variables which it refers to in Γ have different indices in $(j \downarrow \sigma)\,\Gamma$. Concretely, variables $V\,i$ before the point of insertion $(i < j)$ are still correct. But variables with $i \geq j$ need to skip the type σ at j.

In order to implement this, we first define translation of indices.

Definition 5.10. The **up-translation of an index** i **at an insertion point** j is denoted by $i \mapsup j$. Depending on whether i comes before or after j, it is either kept the same or increased by one.

```
_↥_ : (i j : ℕ) → ℕ
_↥_ i j with compare i j
...                | less i<j    :≡ i
...                | equal i=j   :≡ (suc i)
...                | greater i>j :≡ (suc i)
```

◁

This operation can now be extended to terms.

Definition 5.11. The **up-translation of a term** is defined by induction. Constant terms are unaffected. For variables, the up-translation of indices is used. For lambda abstractions, the term inside is translated, but since the lambda introduces a new variable itself, the insertion point j has to be incremented. For applications, both the function and its argument are translated.

```
_⇈_ : Term → ℕ → Term
_⇈_ (cconst x) j :≡ cconst x
_⇈_ (V i)      j :≡ V (i ↥ j)
_⇈_ (Λ σ t)    j :≡ Λ σ (t ⇈ suc j)
_⇈_ (app f x)  j :≡ app (f ⇈ j) (x ⇈ j)
```

◁

The operation of up-translating a term is exactly what is needed when weakening a context. This is stated as a theorem.

Theorem 5.12 (Weakening)**.** *For a term* $\Gamma \vdash t :: \tau$*, well-typedness in a weakened context* $(j \downarrow \sigma)\,\Gamma$ *is achieved by up-translating* t *at* j*.*

```
weak : ∀{τ t} → {Γ : Ctx n}
              → (Γ ⊢ t :: τ)
              → (σ : Ty) → (j : Fin (suc n))
              → (j ↓ σ) Γ ⊢ (t ⇈ o j) :: τ
```

5.7 Substitution

Substitution is the operation of replacing variables in a term with their respective values. First, the operation of simultaneously substituting all variables is encoded in a type. Then, its effect on a term can be stated.

Definition 5.13. A **simultaneous substitution of terms** is encoded as an infinite list of terms, mapping every possible index to a new term.

```
TSub : 𝒰₀
TSub :≡ ℕ → Term
```

◁

Before continuing, we have to consider how substitution is going to work inside of lambda abstractions. A lambda abstraction inserts a new variable at the front of the context, which means that inside, all previous variables are accessed using indices that are incremented by one. It follows that, in order to apply a substitution inside of a lambda, we need to modify it to correctly handle the new variable names.

Definition 5.14. We call such a modification an **extended substitution**. It maps the newly introduced lambda variable to itself. All other variables are mapped to terms in the original substitution (at a decremented index), which need to be up-translated in order to account for the new indexing.

```
extT : TSub → TSub
extT δ zero     :≡ V 0
extT δ (suc n) :≡ (δ n) ⇈ 0
```

◁

Now the action of a simultaneous substitution on a term can be stated.

Definition 5.15. The **action of a simultaneous substitution** δ **on a term** t is defined by induction on t. A constant remains unchanged. A variable is replaced by the corresponding term in δ. For lambda abstractions, the term inside is substituted using extT δ. For applications, the substitution acts on both, the function and its argument.

```
_[_] : Term → TSub → Term
_[_] (cconst x) δ :≡ cconst x
_[_] (V i)       δ :≡ δ i
_[_] (Λ X t)     δ :≡ Λ X (t [ extT δ ])
_[_] (app f x)  δ :≡ app (f [ δ ]) (x [ δ ])
```

◁

Typing information can be added to substitutions. For this, we consider a well-typed term $\Delta \vdash t :: \tau$, to which a substitution δ is going to be applied. Since the variables of t all have to be in Δ, these are the only entries of δ which we have to consider. It is now natural to add the following requirement for δ: All replacements terms must have the same type as the variable which they replace. Additionally, since the replacement terms may contain variables themselves, they all have to be valid in the same context Γ.

Such a typed substitution is called a context morphism:

Definition 5.16. A **context morphism between** Γ **and** Δ is a substitution δ, together with a proof that for every variable in Δ, its replacement term has the same type, as checked in the context Γ.

```
_⇉_ : Ctx m → Ctx n → 𝒰₀
_⇉_ Γ Δ :≡ Σ (λ (δ : TSub) → Π (λ i → Γ ⊢ δ (o̲ i) :: Δ i))
```

◁

For the next step, we add typing information to the extension of substitutions, giving us an extension of context morphisms.

Definition 5.17. The **extension of context morphisms** is defined using the extension of substitions, as well as context weakening (Theorem 5.12). It has the following type:

```
extM : {Γ : Ctx m} → {Δ : Ctx n} → (σ : Ty)
     → (Γ ⇉ Δ)
     → (σ ,, Γ) ⇉ (σ ,, Δ)
```

◁

Now the following theorem about substition can be stated and proven:

Theorem 5.18 (Substitution)**.** *Substituting a well typed term $\Delta \vdash t :: \tau$ with a context morphism $\delta : \Gamma \rightrightarrows \Delta$ preserves well-typedness.*

```
_[_]⇓ : ∀{t σ} → {Γ : Ctx m} → {Δ : Ctx n}
               → Δ ⊢ t :: σ
               → (δ : Γ ⇉ Δ)
               → Γ ⊢ t [ fst δ ] :: σ
```

Proof. This proof works by induction on the term t. For the case of a lambda term, it recursively calls itself with an extended δ (as in Definition 5.17), in order to accomodate for the newly introduced variable. □

5.8 Single substitution

Having defined simultaneous substitution, we can, as a special case of it, define single substitution. Here, only a single variable gets replaced.

Definition 5.19. The **single substitution** of the j-th variable with the term t is denoted by $j \,/\, t$.

It is defined as a simultaneous substitution, where the j-th variable is replaced with t, variables with index $i < j$ are kept the same, and variables with index $i > j$ are down-translated, filling in the hole left at index j. This is done using ↧, which is defined analogously to ↥.

```
_/_ : ℕ → Term → TSub
_/_ j t i with compare–eq i j
_/_ j t i     | equal _ :≡ t
_/_ j t i     | noteq _ :≡ V (i ↧ j)
```

◁

For the case of substituting the first variable, we define a corresponding context morphism.

Definition 5.20. For a well-typed term $T : \Gamma \vdash t :: \tau$, the following context morphism can be defined.

```
Sub0 : ∀{t τ} → {Γ : Ctx m} → (T : Γ ⊢ t :: τ) → Γ ⇉ (τ ,, Γ)
Sub0 {m} {t} {τ} {Γ} T :≡ (0 / t) , proof
  where
    proof : (i : Fin (suc m)) → Γ ⊢ (0 / t) (o̲ i) :: (τ ,, Γ) i
```

Here, the implementation of proof uses T for the case of $i = 0$, and the fact that $\Gamma \vdash \mathsf{V}\, i :: \Gamma\, i$ for $i > 0$. ◁

5.9 Reduction

A $\lambda\!\!\to$ term can be executed. This means that applying a function to an argument is evaluated by substituting the function variable with the argument. Such a process is called β-reduction.

In this section we introduce β-reduction as a relation between terms. The definitions are based on Twelf code found in Schürmann [21].

Definition 5.21. The **single step** β **reduction** of a term t to a term u is denoted by $t \mapsto u$ and defined as the following inductive data type.

```
data _↦_ : Term → Term → 𝒰0 where
  rbeta : ∀{σ r s t u} → (t = app (Λ σ r) s) → (u = r [ 0 / s ]) → t ↦ u
  rlam  : ∀{σ r s}     → r ↦ s → Λ σ r ↦ Λ σ s
  rapp1 : ∀{r s t u v} → (r ↦ s) → (t = app r v) → (u = app s v) → t ↦ u
  rapp2 : ∀{r s t u v} → (r ↦ s) → (t = app v r) → (u = app v s) → t ↦ u
```

◁

The actual reduction is performed in rbeta, where the application of a lambda abstraction $\Lambda\, \sigma\, r$ to a term s is reduced to $r\,[\,0/s\,]$. The other constructors allow for beta reduction to be performed inside lambda terms (rlam) and on both sides of a function application (rapp1 and rapp2).

The constructor rlam differs from the other three in that its output type directly expresses the form of the resulting reduction, while everywhere else the auxilliary terms t and u are used, together with proofs about what form they should have. This is necessitated by the implementation of the normalization proof mentioned in the next section, which needs to be able to explictly transform these equalities. But since this is not necessary for rlam, it can be stated here in this more concise form.

Remark. When considering the connection of type theory and logic, β-reduction corresponds to cut-elimination [12].

Definition 5.22. The **multi step β reduction** of a term t to a term u is denoted by $t \mapsto\!\ast\, u$ and defined as a sequence of single step reductions.

```
data _↦∗_ : Term → Term → 𝓤₀ where
  rid   : ∀{t} → t ↦∗ t
  _•∘_ : ∀{r s t} → r ↦ s → s ↦∗ t → r ↦∗ t
```

◁

Evaluating a term should not change its type. This is formulated as a theorem.

Theorem 5.23. *For every well-typed term t, the term u obtained by a single reduction step is also well-typed.*

```
JStep : ∀{t u τ} → {Γ : Ctx n} → (t ↦ u) → Γ ⊢ t :: τ → Γ ⊢ u :: τ
```

Proof. This proof works by induction on the constructors of reduction. For the case of rbeta, Theorem 5.18 is be used. □

5.10 Normal form

A term is called normal if it cannot be reduced any further [19]. Consequently, an important goal for a type theory is to be normalizing, i.e., for every term in it to have a normal form.

More practically, we want to have a reduction algorithm for λ→ which can be used to evaluate terms. Proving that such an algorithm terminates would imply normalization.

But since in Agda all functions have to be terminating, and a termination proof for an algorithm like stated in Sestoft [23] seems to be to be too complex to be automatically derived by the Agda typechecker, we cannot even define this algorithm without resorting to macros which disable the termination checker.

Because of this, we approach this problem by first proving normalization for terms of λ→. Such a proof, being necessarily constructive, gives us a reduction algorithm for free.

The proof we use is an adaptation of Abel's proof of *Normalization for the Simply-Typed Lambda-Calculus in Twelf* [1]. It is not repeated here - only the necessary definitions and conclusions are stated.

The proof works by taking the well-typedness of the term which is being reduced into account. As a result, the definition of a term in normal form also contains typing information.

Definition 5.24. A term in **typed normal form** is defined by two mutually inductive datatypes.

```
data _⊢_↓_ {n : ℕ} (Γ : Ctx n) : Term → Ty → 𝓤₀
data _⊢_↑_ {n : ℕ} (Γ : Ctx n) : Term → Ty → 𝓤₀
```

We write $\Gamma \vdash t \downarrow \tau$ if t is normal and neutral, i.e., if it is either a variable (ne—var) or a constant (ne—const), or if it is an application (ne—app) where the function term is neutral (and as such does not contain lambda expressions) and therefore cannot be reduced.

```
data _⊢_↓_ {n} Γ where
  ne—var   : (i : Fin n) → {σ : Ty} → (Γ i ≡ σ) → Γ ⊢ V (o̲ i) ↓ σ
  ne—const : (c : Const) → {σ : Ty} → (ι (ctype c) ≡ σ) → Γ ⊢ cconst c ↓ σ
  ne—app   : ∀{r s ρ σ} → (Γ ⊢ r ↓ (σ ⇒ ρ)) → (Γ ⊢ s ↑ σ)
                        → Γ ⊢ (app r s) ↓ ρ
```

We write $\Gamma \vdash t \uparrow \tau$ if t is normal, i.e., if it is either neutral (nf—ne) or a lambda abstraction of a normal term (nf—lam).

```
data _⊢_↑_ {n} Γ where
  nf—ne  : ∀{t τ} → (Γ ⊢ t ↓ τ) → Γ ⊢ t ↑ τ
  nf—lam : ∀{s σ τ ψ} → (ψ ≡ σ ⇒ τ) → ((σ ,, Γ) ⊢ s ↑ τ) → Γ ⊢ Λ σ s ↑ ψ
```

◁

This definition guarantees that a normal term cannot have subterms of the form app $(\Lambda\, \sigma\, t)\; s$. This is because lambda abstractions can only be formed at the outermost level of a term or in the argument position of an application. Without such subterms, no β-reduction can be done, making terms in normal form irreducible.

Lemma 5.25. *A term in typed normal form is well-typed.*

```
nfj↑ : ∀{n t A} → {Γ : Ctx n} → Γ ⊢ t ↑ A → Γ ⊢ t ∷ A
```

Proof. Since the definition of a typed normal form already captures the concept of well-typedness, this proof is trivial. □

Theorem 5.26 (Weak normalization)**.** *For every well-typed term* $\Gamma \vdash t :: \tau$ *there exists a sequence of reduction steps* $t \mapsto *u$ *to a term in normal form* $\Gamma \vdash u \uparrow \tau$*.*

```
nf : ∀{t τ} → {Γ : Ctx n} → Γ ⊢ t ∷ τ → Σ (λ u → (t ↦∗ u) × (Γ ⊢ u ↑ τ))
```

Proof. Omitted. □

Remark. This is a proof of weak normalization. Strong normalization would imply that *every* sequence of reduction steps terminates.

The normalization algorithm for $\lambda{\rightarrow}$ is defined as follows:

Definition 5.27. The **normal form** of a well-typed term t is the term u whose existence was proven above.

```
nor : ∀{t τ} → {Γ : Ctx n} → Γ ⊢ t ∷ τ → Term
nor T :≡ fst (nf T)
```

◁

Using this, a notion of equality between terms can be introduced:

Definition 5.28. Two terms are said to be **β-equal** if their normal forms are equal.

```
_=β=_ : ∀{t u τ} → {Γ : Ctx n} → (Γ ⊢ t :: τ) → (Γ ⊢ u :: τ) → 𝒰₀
_=β=_ T U :≡ nor T = nor U
```

◁

Remark. This is the concept behind the definitional equality (≡) which was mentioned in chapter 2.

5.11 Example: Church numerals

As an example of working in $\lambda{\to}$, we present a small formalization of natural numbers, using *Church numerals* [22].

We choose Bool as the type of ground types and also as the type of constants, mapped via the identity function.

```
param : LambdaParam i j
param :≡ lambdaParam Bool Bool–isDisc Bool idf
```

We call the two resulting types α and β, and their only terms a and b respectively.

```
α β : Ty
α :≡ ι true
β :≡ ι false

a b : Term
a :≡ cconst true
b :≡ cconst false
```

For better legibility when constructing terms, we introduce the following notation:

```
_⇨_ :≡ Λ
_$_  :≡ app
```

Since the normalization algorithm only works on well-typed terms, we do not work directly with terms, but with proofs of their well-typedness. Therefore, we also introduce a shorter notation for applying a well-typed function to a well-typed term:

```
_#_ :≡ app⇓
```

Now the identity function on α can be written as follows:

```
idα : [] ⊢ α ⇨ V 0 :: α ⇒ α
idα :≡ tt ,fir, refl
```

And, given a well-typed constant aa:

```
aa : [] ⊢ a :: α
aa :≡ tt , fir, refl
```

We can show that applying the identity function to it does nothing:

```
th1 : idα # aa =β= aa
th1 :≡ refl
```

The natural numbers are defined as higher order functions. The n-th numeral is encoded as the function which maps a function f to its n-times repeated composition with itself, $f \mapsto f^n$. We call this type NN.

```
NN : Ty
NN :≡ (α ⇒ α) ⇒ (α ⇒ α)
```

The first 4 natural numbers are defined as follows:

```
n0 : [] ⊢ (α ⇒ α) ⇨ (α) ⇨ V 0 :: NN
n1 : [] ⊢ (α ⇒ α) ⇨ (α) ⇨ V 1 $ V 0 :: NN
n2 : [] ⊢ (α ⇒ α) ⇨ (α) ⇨ V 1 $ (V 1 $ V 0) :: NN
n3 : [] ⊢ (α ⇒ α) ⇨ (α) ⇨ V 1 $ (V 1 $ (V 1 $ V 0)) :: NN
```

They all typecheck correctly:

```
n0 :≡ tt , fir, refl
n1 :≡ tt , fir, refl
n2 :≡ tt , fir, refl
n3 :≡ tt , fir, refl
```

We can define a successor function:

```
nsuc : [] ⊢ NN ⇨ (α ⇒ α) ⇨ α ⇨ V 1 $ (V 2 $ V 1 $ V 0) :: NN ⇒ NN
nsuc :≡ tt , fir, refl
```

And check that $\mathsf{suc}(\mathsf{suc}\,0) = 2$.

```
th2 : nsuc # (nsuc # n0) =β= n2
th2 :≡ refl
```

We can also define addition and multiplication:

```
++ : [] ⊢ NN ⇨ NN ⇨ (α ⇒ α) ⇨ α ⇨ (V 3 $ V 1) $ (V 2 $ V 1 $ V 0)
          :: NN ⇒ NN ⇒ NN
++ :≡ tt , fir, refl

** : []  ⊢ NN ⇨ NN ⇨ (α ⇒ α) ⇨ α ⇨ (V 3 $ (V 2 $ V 1) $ V 0)
          :: NN ⇒ NN ⇒ NN
** :≡ tt , fir, refl
```

And test their properties.

```
th3 : (++ # n1 # n2) =β= n3
th3 :≡ refl
```

```
th4 : (** # n0 # n1) =β= n0
th4 :≡ refl

th5 : (** # n2 # n3) =β= (++ # n3 # n3)
th5 :≡ refl
```

Remark. Here, # is used to apply the functions ++ and ** to two arguments.

All of these correct statements typecheck, while false propositions would not. This shows that $\lambda\!\rightarrow$ is powerful enough to encode basic arithmetic.

6 Interpretation

In this chapter we present categorical semantics for the simply typed λ-calculus by giving an interpretation into cartesian closed categories. The definitions and the structure of the theorems, as well as the core ideas for proving them are taken from Pitts [20].

6.1 Parametrization

In order to formulate an interpretation, we need to choose a parametrization for $\lambda{\to}$, as well as a corresponding CCC into which it can be interpreted. Such a choice is again encoded in a parametrization.

Definition 6.1. A parametrization of the interpretation is given by the following record:

```
record IParam (i j : ULevel) : 𝒰 (lsuc (lmax i j)) where
  constructor iParam
```

It contains a parametrization of $\lambda{\to}$,

```
  field
    param : LambdaParam i j
```

a cartesian closed category $\mathcal{C}$,

```
  field
    𝒞   : Category i j
    CCC : isCCC 𝒞
```

a function M, relating ground types of $\lambda{\to}$ to objects in $\mathcal{C}$, and a function Mc, relating constants to global sections of their respective type.

```
  field
    M  : Gnd → Obj
    Mc : (c : Const) → 𝟏 ⇁ (M (ctype (c)))
```

◁

Remark. A global section of an object A is simply a morphism $1 \rightharpoondown A$.

6.2 Definition

The interpretation is divided into four seperate functions: the interpretation of types (T⟦_⟧), of contexts (C⟦_⟧), of typing judgements (J⟦_⟧) and of context morphisms

$(\mathsf{M}[\![\_]\!])$.

Definition 6.2. A type of $\lambda{\to}$ is interpreted as an object of $\mathcal{C}$. For ground types, M is used. Function types are mapped to exponential objects.

$$
\begin{aligned}
&\mathsf{T}[\![\_]\!] : \mathsf{Ty} \to \mathsf{Obj} \\
&\mathsf{T}[\![\_]\!]\ (\iota\ x) \qquad :\equiv \mathsf{M}\ x \\
&\mathsf{T}[\![\_]\!]\ (A \Rightarrow B) :\equiv \mathsf{T}[\![\ B\ ]\!]\ \hat{}\ \mathsf{T}[\![\ A\ ]\!]
\end{aligned}
$$

◁

Definition 6.3. A context of $\lambda{\to}$ is interpreted as the finite product of its types (themselves interpreted first).

$$
\begin{aligned}
&\mathsf{C}[\![\_]\!] : \forall\{n\} \to \mathsf{Ctx}\ n \to \mathsf{Obj} \\
&\mathsf{C}[\![\_]\!]\ \Gamma :\equiv \textstyle\prod\ (\mathsf{T}[\![\_]\!] \circ \Gamma)
\end{aligned}
$$

◁

Remark. Similarly to $\mathsf{O}{=}[\![\_]\!]$, which turns equalities of objects into arrows, we define $\mathsf{T}{=}[\![\_]\!]$ and $\mathsf{C}{=}[\![\_]\!]$ for equalities of types and of contexts.

Definition 6.4. A typing judgement $\Gamma \vdash t :: \tau$ is interpreted as a morphism from the context $\mathsf{C}[\![\,\Gamma\,]\!]$ to the type $\mathsf{T}[\![\,\tau\,]\!]$:

$$
\mathsf{J}[\![\_]\!] : \forall\{t\ \tau\} \to \{\Gamma : \mathsf{Ctx}\ n\} \to (\Gamma \vdash t :: \tau) \to \mathsf{C}[\![\ \Gamma\ ]\!] \rightharpoondown \mathsf{T}[\![\ \tau\ ]\!]
$$

(i) A constant term $\mathsf{cconst}\,c$ is interpreted using the terminal arrow ! and the global section $\mathsf{Mc}\,c$. Finally, since the target type of the resulting arrow must be $\mathsf{T}[\![\,\tau\,]\!]$, a type correction has to be added using $\mathsf{T}{=}[\![\_]\!]$.

$$
\mathsf{C}[\![\,\Gamma\,]\!] \xrightarrow{\ !\ } \mathbf{1} \xrightarrow{\mathsf{Mc}\,c} \mathsf{M}\ (\mathsf{ctype}\ c) \xrightarrow{\mathsf{T}=[\![\,p\,]\!]} \mathsf{T}[\![\,\tau\,]\!]
$$

$$
\begin{aligned}
\mathsf{J}[\![\_]\!]\ \{t :\equiv \mathsf{cconst}\ c\}\ T :\equiv\ &\mathsf{let}\ p :\equiv \mathsf{cconst}{\Uparrow}\ T \\
&\mathsf{in}\ \ !\diamond (\mathsf{Mc}\ c) \diamond \mathsf{T}{=}[\![\ p\ ]\!]
\end{aligned}
$$

(ii) A variable with index i is interpreted by the i-th projection arrow $\pi_i\,i$, followed by a type correction.

$$
\mathsf{C}[\![\,\Gamma\,]\!] \xrightarrow{\pi_i\ i} \mathsf{T}[\![\,\Gamma\,i\,]\!] \xrightarrow{T=[\![\Gamma i=\tau]\!]} \mathsf{T}[\![\,\tau\,]\!]
$$

$$
\begin{aligned}
\mathsf{J}[\![\_]\!]\ \{t :\equiv \mathsf{V}\ x\}\ \{\tau\}\ T :\equiv\ &\mathsf{let}\ i\ ,\ \Gamma i{=}\tau\ ,\ \_ :\equiv \mathsf{V}{\Uparrow}\ T \\
&\mathsf{in}\ \ \pi_{\mathsf{i}}\ i \diamond \mathsf{T}{=}[\![\ \Gamma i{=}\tau\ ]\!]
\end{aligned}
$$

(iii) A lambda abstraction $\Lambda\sigma\mathsf{r}$ is interpreted recursively: Since its type has to be a function type $(\psi \Rightarrow \rho)$, we can use $\Lambda\ {\Uparrow}{\Rightarrow}$ to get a judgement $(\psi, , \Gamma) \vdash r :: \rho$. Interpreting this, we get a morphism $\mathsf{C}[\![\,\Gamma\,]\!] \times \mathsf{T}[\![\,\psi\,]\!] \rightharpoondown \mathsf{T}[\![\,\rho\,]\!]$, which we can curry to get a morphism $\mathsf{C}[\![\,\Gamma\,]\!] \rightharpoondown \mathsf{T}[\![\,\rho\,]\!]\,\hat{}\,\mathsf{T}[\![\,\psi\,]\!]$. A type correction has to be added.

$$\mathsf{C}[\![\,\Gamma\,]\!] \xrightarrow{\mathsf{C{=}}[\![\,\mathsf{tail{=}}\;\psi\;\Gamma\,]\!]} \mathsf{C}[\![\,\mathsf{tail}\,(\psi, , \Gamma)\,]\!] \xrightarrow{\mathsf{curry}\;\mathsf{J}[\![\,R\,]\!]} \mathsf{T}[\![\,\rho\,]\!]\,\hat{}\,\mathsf{T}[\![\,\psi\,]\!] = \mathsf{T}[\![\,\tau\,]\!]$$

```
J⟦_⟧ {t :≡ Λ σ r} {ι _}           ΛR :≡ ⊥–elim (Λ⇑ι ΛR)
J⟦_⟧ {t :≡ Λ σ r} {ψ ⇒ ρ} {Γ} ΛR :≡ let R , _ :≡ Λ⇑⇒ ΛR
                                    in  C=⟦ tail= ψ Γ ⟧ ⋄ curry J⟦ R ⟧
```

(iv) An application $\mathsf{app}\;t\;s$ is also interpreted recursively: The typing judgements for t and s are interpreted individually, resulting in the morphisms $\mathsf{C}[\![\,\Gamma\,]\!] \rightharpoondown \mathsf{T}[\![\,\tau\,]\!]\hat{}\mathsf{T}[\![\,\sigma\,]\!]$ and $\mathsf{C}[\![\,\Gamma\,]\!] \rightharpoondown \mathsf{T}[\![\,\sigma\,]\!]$. These are combined using the product of morphisms and then joined with ev.

$$\mathsf{C}[\![\,\Gamma\,]\!] \xrightarrow{\langle \mathsf{J}[\![\,T\,]\!]\,,\,\mathsf{J}[\![\,S\,]\!]\rangle} \mathsf{T}[\![\,\tau\,]\!]\,\hat{}\,\mathsf{T}[\![\,\sigma\,]\!] \times \mathsf{T}[\![\,\sigma\,]\!] \xrightarrow{\qquad\mathsf{ev}\qquad} \mathsf{T}[\![\,\tau\,]\!]$$

```
J⟦_⟧ {t :≡ app t s} {τ} TS :≡ let σ , T , S :≡ app⇑ TS
                               in  ⟨ J⟦ T ⟧ , J⟦ S ⟧ ⟩ ⋄ ev
```

◁

Definition 6.5. A context morphism is interpreted as a finite product over the interpretations of the judgements it contains.

```
M⟦_⟧ : {Γ : Ctx m} → {Δ : Ctx n} → (f : Γ ⇉ Δ) → C⟦ Γ ⟧ ⇁ C⟦ Δ ⟧
M⟦_⟧ (f , F) :≡ ⟪ (λ i → J⟦ F i ⟧) ⟫
```

◁

6.3 Properties

Having defined the interpretation functions, we can now state how they interact with concepts like weakening and substitution.

Lemma 6.6. *The context morphism of substituting the first variable with a term* $T : \Gamma \vdash t :: \tau$ *is like the product of* id *and* $\mathsf{J}[\![\,T\,]\!]$, *except that a type correction arrow is used instead of* id. *Using diagrams, we say that the arrow*

$$\mathsf{C}[\![\,\Gamma\,]\!] \xrightarrow{\qquad\mathsf{M}[\![\,\mathsf{Sub}_0\;T\,]\!]\qquad} \mathsf{C}[\![\,\tau, , \Gamma\,]\!]$$

is equal to the following:

$$\mathsf{C}[\![\,\Gamma\,]\!] \xrightarrow{\langle \mathsf{C{=}}[\![\,\mathsf{tail{=}}\;\tau\;\Gamma\,]\!]\,,\,\mathsf{J}[\![\,T\,]\!]\rangle} \mathsf{C}[\![\,\mathsf{tail}\,(\tau, , \Gamma)\,]\!] \times \mathsf{T}[\![\,\tau\,]\!]$$

In Agda this is formalized using the following statement:

```
lSub₀ : ∀{t τ} → {Γ : Ctx m}
               → (T : Γ ⊢ t ∷ τ)
               → M⟦ Sub₀ T ⟧ = ⟨ C=⟦ (tail= τ Γ) ⟧ , J⟦ T ⟧ ⟩
```

Theorem 6.7 (Semantics of weakening)**.** *The interpretation of a weakened term* weak$T\sigma j$ *is equal to the morphism of type* $\mathsf{C}[\![\,(j\downarrow\sigma)\,\Gamma\,]\!]\to\mathsf{C}[\![\,\Gamma\,]\!]$ *which projects all types except the j-th, followed by* $\mathsf{J}[\![\,T\,]\!]$.

$$\begin{tikzcd}
\mathsf{C}[\![\,(j\downarrow\sigma)\,\Gamma\,]\!] \arrow[r, "\langle\!\langle(\lambda i\to\pi_i(i\,\underline{\uparrow}\mathsf{f}\,j))\rangle\!\rangle"] \arrow[ddr, "\mathsf{J}[\![\,\mathsf{weak}\,T\,\sigma\,j\,]\!]"'] & \mathsf{C}[\![\,\lambda i\to(j\downarrow\sigma)\Gamma(i\,\underline{\uparrow}\mathsf{f}\,j)\,]\!] \arrow[d, "\mathsf{C}{=}[\![\,\mathsf{insertLShiftL}\,\Gamma\,j\,\sigma\,]\!]"] \\
 & \mathsf{C}[\![\,\Gamma\,]\!] \arrow[d, "\mathsf{J}[\![\,T\,]\!]"] \\
 & \mathsf{T}[\![\,\tau\,]\!]
\end{tikzcd}$$

```
lWeak : ∀{m t τ} → {Γ : Ctx m} → (σ : Ty)
                 → (T : Γ ⊢ t ∷ τ) → (j : Fin (suc m))
                 → J⟦ weak T σ j ⟧
                   =
                  ⟪ (λ i → πᵢ {A ≔ λ k → T⟦ ((j ↓ σ) Γ) k ⟧} (i ↥f j)) ⟫
                  ◇ C=⟦ insertLShiftL Γ j σ ⟧
                  ◇ J⟦ T ⟧
```

Remark. The function _↥f_ is like _↥_, but defined for finite indices instead of natural numbers. The term insertLShiftL $\Gamma\,\sigma\,j$ is a proof of

$$(\lambda i\to(j\downarrow\sigma)\,\Gamma\,(i\,\underline{\uparrow}\mathsf{f}\,j))=\Gamma$$

meaning that inserting an element into a list Γ, and then building a list which skips this element is equal to the original list.

Corollary 6.8. *The interpretation of weakening can be specialized to the case where an element is inserted at the front. Instead of the complex projection function which skips the j-th object from before, we can simply use π_1, projecting the tail of $(0\downarrow\sigma)\,\Gamma$.*

$$\begin{tikzcd}
\mathsf{C}[\![\,(0\downarrow\sigma)\,\Gamma\,]\!] \arrow[r, "\pi_1"] \arrow[ddr, "\mathsf{J}[\![\,\mathsf{weak}\,T\,\sigma\,0\,]\!]"'] & \mathsf{C}[\![\,\mathsf{tail}((0\downarrow\sigma)\,\Gamma)\,]\!] \arrow[d, "\mathsf{C}{=}[\![\,\mathsf{sym}(\mathsf{tail}{=}\,\sigma\,\Gamma)\,]\!]"] \\
 & \mathsf{C}[\![\,\Gamma\,]\!] \arrow[d, "\mathsf{J}[\![\,T\,]\!]"] \\
 & \mathsf{T}[\![\,\tau\,]\!]
\end{tikzcd}$$

$$\begin{aligned}\mathsf{lWeak}_0 : \forall\{t\ \tau\} \to \{\Gamma : \mathsf{Ctx}\ m\} \to (\sigma : \mathsf{Ty}) \to (T : \Gamma \vdash t :: \tau)\\ \to \mathsf{J}\llbracket\ \mathsf{weak}\ T\ \sigma\ \mathsf{fzero}\ \rrbracket\\ =\\ \pi_1 \diamond \mathsf{C}{=}\llbracket\ \mathsf{sym}\ (\mathsf{tail}{=}\ \sigma\ \Gamma)\ \rrbracket \diamond \mathsf{J}\llbracket\ T\ \rrbracket\end{aligned}$$

Proof. This statement is a special case of Theorem 6.9. □

Remark. Here, $\mathsf{tail}{=}$ can be used since the definitional equality $\mathsf{tail}\ ((\mathsf{fzero} \downarrow \sigma)\ \Gamma) \equiv \mathsf{tail}\ (\sigma, , \Gamma)$ holds.

Lemma 6.9 (Semantics of extending a context morphism)**.** *The arrow*

$$\mathsf{C}\llbracket\, \sigma, , \Gamma\, \rrbracket \xrightarrow{\mathsf{M}\llbracket\, \mathsf{extM}\ \sigma\ F\, \rrbracket} \mathsf{C}\llbracket\, \sigma, , \Delta\, \rrbracket$$

can be split into the arrows

$$\mathsf{C}\llbracket\, \mathsf{tail}\ (\sigma, , \Gamma)\, \rrbracket \xrightarrow{\mathsf{C}{=}\llbracket\, \mathsf{sym}\ (\mathsf{tail}{=}\ \sigma\ \Gamma)\, \rrbracket \diamond \mathsf{M}\llbracket\, F\, \rrbracket \diamond \mathsf{C}{=}\llbracket\, \mathsf{tail}{=}\ \sigma\ \Delta\, \rrbracket} \mathsf{C}\llbracket\, \mathsf{tail}\ (\sigma, , \Delta)\, \rrbracket$$

and

$$\mathsf{T}\llbracket\, \sigma\, \rrbracket \xrightarrow{\qquad\qquad\qquad \mathsf{id} \qquad\qquad\qquad} \mathsf{T}\llbracket\, \sigma\, \rrbracket$$

$$\begin{aligned}\mathsf{lext} : \{\Gamma : \mathsf{Ctx}\ m\} \to \{\Delta : \mathsf{Ctx}\ n\} \to (F : \Gamma \rightrightarrows \Delta) \to (\sigma : \mathsf{Ty})\\ \to \mathsf{M}\llbracket\ \mathsf{extM}\ \sigma\ F\ \rrbracket\\ =\\ (\mathsf{C}{=}\llbracket\ \mathsf{sym}\ (\mathsf{tail}{=}\ \sigma\ \Gamma)\ \rrbracket \diamond \mathsf{M}\llbracket\ F\ \rrbracket \diamond \mathsf{C}{=}\llbracket\ \mathsf{tail}{=}\ \sigma\ \Delta\ \rrbracket) \times\times\ \mathsf{id}\end{aligned}$$

Proof. This proof uses Corollary 6.8 in order to decompose arrows of type $\mathsf{C}\llbracket\, \sigma, , \Gamma\, \rrbracket \to \mathsf{T}\llbracket\, \Delta\ i\, \rrbracket$ into π_1 and an arrow of type $\mathsf{C}\llbracket\, \Gamma\, \rrbracket \to \mathsf{T}\llbracket\, \Delta\ i\, \rrbracket$. □

Theorem 6.10 (Semantics of substitution)**.** *The interpretation of a substitution $T\,[\,F\,] \Downarrow$ is a composition of the interpretations of the context morphism F and the judgement T.*

$$\begin{tikzcd}
\mathsf{C}\llbracket\, \Delta\, \rrbracket \arrow[r, "\mathsf{M}\llbracket\, F\, \rrbracket"] \arrow[dr, "\mathsf{J}\llbracket\, T\,[\,F\,]\Downarrow\, \rrbracket"'] & \mathsf{C}\llbracket\, \Gamma\, \rrbracket \arrow[d, "\mathsf{J}\llbracket\, T\, \rrbracket"] \\
 & \mathsf{T}\llbracket\, \tau\, \rrbracket
\end{tikzcd}$$

$$\begin{aligned}\mathsf{lSub} : \forall\{t\ \tau\} \to \{\Gamma : \mathsf{Ctx}\ m\} \to \{\Delta : \mathsf{Ctx}\ n\}\\ \to (T : \Gamma \vdash t :: \tau)\\ \to (F : \Delta \rightrightarrows \Gamma)\\ \to \mathsf{J}\llbracket\ T\ [\ F\ ]\Downarrow\ \rrbracket = \mathsf{M}\llbracket\ F\ \rrbracket \diamond \mathsf{J}\llbracket\ T\ \rrbracket\end{aligned}$$

Proof. Similar to the proof of Theorem 5.18, this proof uses Lemma 6.9 for the case of t being a lambda term. □

6.4 Soundness

The interpretation of λ→ terms into categories should be compatible with the internal notion of β-equality: Terms which are considered equal should have the same interpretation. Such a property is called soundness.

β-equality is based on reduction, therefore the main challenge is to prove that a single reduction step does not change the interpretation of a term.

Theorem 6.11. *The interpretation of a well typed term does not change after a single reduction step.*

```
SingleStep : ∀{t u τ} → {Γ : Ctx n}
                      → (w : t ↦ u)
                      → (T : Γ ⊢ t :: τ)
                      → J⟦ T ⟧ = J⟦ JStep w T ⟧
```

Proof. The proof works by induction on the definition of a single reduction step. The most interesting case is that of rbeta, it involves substition of the first variable. In order to prove it, we have to use the properties described in Lemma 6.6 and Theorem 6.10. □

By combining multiple steps, and then applying the resulting proof to the case of normalization, the following two corollaries are obtained.

Corollary 6.12. *The interpretation of a well typed term does not change after multiple reduction steps.*

```
MultiStep : ∀{t u τ} → {Γ : Ctx n}
                     → (w : t ↦∗ u)
                     → (T : Γ ⊢ t :: τ) → (U : Γ ⊢ u :: τ)
                     → J⟦ T ⟧ = J⟦ U ⟧
```

Corollary 6.13. *The interpretation of a term and of its normal form are the same.*

```
norsound : ∀{t τ} → {Γ : Ctx n}
                  → (T : Γ ⊢ t :: τ)
                  → J⟦ T ⟧ = J⟦ nor⇓ T ⟧
```

Finally, this can be used to show soundness.

Corollary 6.14 (Soundness)**.** *The interpretation is sound with respect to β-equality.*

```
sound : ∀{t u τ} → {Γ : Ctx n}
                 → (T : Γ ⊢ t :: τ) → (U : Γ ⊢ u :: τ)
                 → (T =β= U)
                 → J⟦ T ⟧ = J⟦ U ⟧
```

Proof. Since the normal forms $\mathsf{nor}\Downarrow T$ and $\mathsf{nor}\Downarrow U$ are equal, so are their interpretations:

$$\mathsf{J}[\![\, \mathsf{nor}\Downarrow T \,]\!] = \mathsf{J}[\![\, \mathsf{nor}\Downarrow U \,]\!]$$

By applying Corollary 6.13 to both sides, this means that the interpretations of the original terms have to be equal as well:

$$\mathsf{J}[\![\, T \,]\!] = \mathsf{J}[\![\, U \,]\!]$$ □

7 Discussion and further work

This shows how an implementation of $\lambda\!\rightarrow$ and its interpretation into a CCC can be formalized in Agda.

Nevertheless, some aspects of the code are not entirely satisfactory:

- The definitions of finite types and finite lists are problematic. Their encoding was chosen to fit the problems encountered early on. But in chapter 6, this resulted in many correction arrows having to be added, making theorems and proofs needlessly more complex. Also, because of how these arrows are defined, the interpretation function for judgements does not compute.

  These problems could have been avoided with a recursive definition of lists - but this in turn would produce its own set of problems, presumably around the definition of weakening, since there, arbitrary indices into a list have to be dealt with.

  Such a conflict arises because a single definition is used everywhere, and it could be solvable if different definitions which fundamentally describe the same object could be used interchangebly.

  Fortunately, this seems to be exactly what the concept of univalence provides and it would be interesting to see how it can be applied here.

- The code was only written with focus on correctness, not on performance. Because of this, evaluating the β-equality of simple terms may sometimes take multiple minutes, while using several gigabytes of memory.

  Since this evaluation is done as part of typechecking a file, using the CTT normalization of Agda, it is not immediately clear what the reason for such performance problems could be.

  Still, before any practical usage can happen, this direction has to be explored as well.

Concerning the goal of visualizing programs, this exploration of the semantics of $\lambda\!\rightarrow$ reinforces the idea that category theory and diagrams should be useful tools: it is natural to think about computations as morphisms from a context to their result type.

But it also raises questions: Arrows have no clear representation. As encoded by the very concept of a commutating diagram, the same computation could be visualized using different combinations of arrows. This indicates that there is no canonical way to represent a program diagrammatically, while at the same time, it could be possible to leverage this fact for zooming in and out of a representation.

Finally, $\lambda\!\rightarrow$ is a very simple type theory, missing many of the constructions found in real world programming languages (sum types, general data types, dependent types). Because of this, the next natural step would be to extend it with these concepts, while also moving to appropriate categories, where the corresponding semantics can be formulated.

Bibliography

[1] Andreas Abel. Normalization for the simply-typed lambda-calculus in twelf. *Electronic Notes in Theoretical Computer Science*, 199:3 – 16, 2008. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2007.11.009. URL `http://www.sciencedirect.com/science/article/pii/S1571066108000753`. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

[2] Agda cubical library. URL `https://github.com/Saizan/cubical-demo`. Accessed: 2018-04-08.

[3] Agda online documentation. URL `https://agda.readthedocs.io/en/v2.5.4.1/`. Accessed: 2018-07-27.

[4] Agda prelude library. URL `https://github.com/UlfNorell/agda-prelude`. Accessed: 2018-05-24.

[5] Agda source code repository. URL `https://github.com/agda/agda`. Accessed: 2018-07-27.

[6] Steve Awodey. *Category Theory*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2010. ISBN 0199237182, 9780199237180.

[7] Felice Cardone and J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817, 2006.

[8] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016.

[9] Robert L. Constable. The triumph of types: Principia mathematica's impact on computer science. URL `http://hdl.handle.net/1813/28696`. Accessed: 2018-07-28.

[10] Thierry Coquand and Christine Paulin. Inductively defined types. In *Proceedings of the International Conference on Computer Logic*, COLOG '88, pages 50–66, London, UK, UK, 1990. Springer. ISBN 3-540-52335-9. URL `http://dl.acm.org/citation.cfm?id=646125.758641`.

[11] J. Dunfield and N. R. Krishnaswami. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types, January 2016. arXiv:1601.05106.

[12] Herman Geuvers. Introduction to type theory. In *Language Engineering and Rigorous Software Development*, pages 1–56. Springer, Berlin, Heidelberg, 2009.

[13] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. Types in logic and mathematics before 1940. *Bull. Symbolic Logic*, 8(2):185–245, 06 2002. doi: 10.2178/bsl/1182353871. URL `https://doi.org/10.2178/bsl/1182353871`.

[14] Elaine Landry and Jean-Pierre Marquis. Categories in context: Historical, foundational, and philosophical. *Philosophia Mathematica*, 13(1):1–43, 2005. doi: 10.1093/philmat/nki005. URL `http://dx.doi.org/10.1093/philmat/nki005`.

[15] Per Martin-Löf. An intuitionistic theory of types. *Twenty-five years of constructive type theory*, 36:127–172, 1998.

[16] Anders Mörtberg. Standalone implementation of cubical type theory. URL `https://github.com/mortberg/cubicaltt`. Accessed: 2018-03-18.

[17] nLab authors. database of categories. `http://ncatlab.org/nlab/show/database%20of%20categories`, August 2018. Revision 27. Accessed: 2018-08-13.

[18] nLab authors. dependent type. `http://ncatlab.org/nlab/show/dependent%20type`, August 2018. Revision 15. Accessed: 2018-08-04.

[19] nLab authors. normal form. `http://ncatlab.org/nlab/show/normal%20form`, August 2018. Revision 2. Accessed: 2018-08-04.

[20] Andrew Pitts. Brief notes on the category theoretic semantics of simply typed lambda calculus. URL `https://www.cl.cam.ac.uk/teaching/1617/L108/catl-notes.pdf`. Accessed: 2018-06-06.

[21] Carsten Schürmann. *Automating the meta theory of deductive systems.* PhD thesis, Pittsburgh, PA, USA, 2000.

[22] Peter Selinger. Lecture notes on the lambda calculus, April 2008. arXiv:0804.3434.

[23] Peter Sestoft. Demonstrating lambda calculus reduction. In *The essence of computation*, pages 420–435. Springer, 2002.

[24] Michael Shulman, 2018. URL `https://homotopytypetheory.org/2017/09/16/a-hands-on-introduction-to-cubicaltt/#comment-108334`. Comment on blogpost. Accessed: 2018-07-29.

[25] Micheal Shulman. Homotopical trinitarianism: A perspective on homotopy type theory. Talk, 2018. URL `https://home.sandiego.edu/~shulman/papers/trinity.pdf`. Accessed: 2018-07-28.

[26] Peter Smith. Category theory: A gentle introduction, February 2016. URL `https://www.logicmatters.net/resources/pdfs/GentleIntro.pdf`. Accessed: 2018-07-28.

[27] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[28] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52, Berlin, Heidelberg, 1995. Springer. URL `http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf`. Accessed: 2018-08-05.

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 13. August 2018

Maxim Urschumzew