Saarland University
Faculty of Mathematics and Computer Science

Bachelor's Thesis

Strictly Positive Types in Homotopy Type Theory

Author:
Felix Rech

Advisor:
Steven Schäfer

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Holger Hermanns

Submitted: 24th April 2017

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath:

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent:

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 24th April, 2017

Abstract

We adapt the work of Abbott et al. [1] to construct strictly positive types in Homotopy Type Theory. For this construction we use the concept of containers and show that containers are closed under all strictly positive type formers. Moreover we show that propositional resizing can be used to refine a construction of M-types by Ahrens et al. [2] such that the computation rule for corecursion holds judgmentally. Finally we present a construction of W-types from natural numbers with propositional resizing. Together this leads to the result that we have nested inductive and coinductive types in every type theory that contains dependent functions, dependent pairs, natural numbers, paths, a hierarchy of univalent universes and propositional resizing.

Acknowledgments

I am very grateful to my advisor Steven Schäfer who not only introduced me to this fascinating topic but spent a lot of time discussing my work and helped me with countless valuable ideas. I also want to thank Fabian and Julian for proofreading unfinished drafts of my work. Finally I want to thank Prof. Smolka and Prof. Hermanns for reviewing this thesis.

Contents

1 Introduction

This work presents a construction of nested inductive and coinductive types in *Homotopy Type Theory* [4] (HoTT), which means in particular that we do proof relevant work on equalities. Furthermore, we base our work on a small subset of the theory that is usually assumed as a basis for HoTT. Most notably we need no inductive or coinductive types except natural numbers. In the course of our construction we show that a wide class of types can be represented in a certain normal form. This normal form itself has other useful applications, for example in generic programming, which is the programming of functions that work on all possible data structures.

The inductive and coinductive types that we focus on, are described by the grammar of *strictly positive types*:

$$A, B ::= K \mid x \mid A \times B \mid A + B \mid K \to A \mid \mu\, x.A \mid \nu\, x.A.$$

Here K stands for a constant type and x is a variable. We also have products $(A \times B)$, coproducts $(A + B)$ and function types with constant domain $(K \to A)$. The type formers $\mu\, x.A$ and $\nu\, x.A$ describe the inductive and coinductive types that are fixed points of A as a function in x. For example the inductive type of lists containing natural numbers $\mathsf{List}(\mathbb{N})$ satisfies the equivalence $\mathsf{List}(\mathbb{N}) \simeq (\mathbb{N} \times \mathsf{List}(\mathbb{N})) + \mathbf{1}$, where $\mathbf{1}$ stands for the one-element type. Hence it is equivalent to the inductive fixed point $\mu\, x.(\mathbb{N} \times x) + \mathbf{1}$. The coinductive fixed point $\nu\, x.(\mathbb{N} \times x) + \mathbf{1}$ on the other hand is the type of potentially infinite lists over $\mathbb{N}$. We use the term *strictly positive* type because we allow only constant types on the left side of an arrow. Such a restriction is necessary to guarantee that both fixed points always exist. In general this is not the case. The function $\lambda x.\, x \to \mathbf{0}$ for example with $\mathbf{0}$ as the empty type does not have a fixed point.

To construct strictly positive types, we follow an approach by Abbott et al. [1] and employ the concept of *containers* as a polynomial-like normal form of strictly positive types with a free variable. A container $S \rhd P$ consists of a type of shapes S and a function $P : S \to \mathcal{U}$, that assigns a type of positions from a type universe $\mathcal{U}$ to each shape. Such a container represents the type $\sum_{(s:S)} P(s) \to x$, where x is the free variable. That means, every element consists of a shape and an assignment of a component in x to each position. For example the type $\mathsf{List}(\mathbb{N})$ has representation $\mathbb{N} \rhd \mathsf{Fin}$, where $\mathsf{Fin}(n)$ is the finite type with n elements. The shape tells us the length of a list while the number of positions is exactly the number of components in the list. Containers are useful because they always have inductive and coinductive fixed points that we call W- and M-types. Containers can be extended to model types with multiple free variables. They are closed under all strictly positive type formers, which allows us to build the container representation of a strictly positive type step by step.

Abbott et al. [1] showed that in a type theory with uniqueness of identity proofs, M-types can be reduced to natural numbers. We can think of inhabitants of an M-type as trees with potentially unlimited depth. The reduction represents such a tree as a series of finite approximations with increasing depth. Ahrens et al. [2] showed that this construction can also be executed in HoTT. However, the computation rule for corecursion holds only as a propositional equality, not judgmentally. This problem can be overcome, if we are willing to extend the core theory by *propositional resizing*. Roughly speaking this allows us to assume that every type with no more than one inhabitant lives in the smallest universe. We can use it to refine our M-type such that the computation rule holds by definition.

Like M-types, we can also reduce W-types to natural numbers. The intuition of M-types as types of trees applies to W-types as well, with the difference that all elements of a W-type have to be well-founded. This means that we can define every W-type as subtype of the corresponding M-type that contains only well-founded trees. Well-foundedness can be defined as the fact that a weak form of the induction principle for W-types holds on a given element. As with M-types there is a way to add some judgmental equalities to a W-type: We can get a judgmental computation rule for simple recursion if every element carries its own recursor.

Finally we can reduce our core theory even further by dropping some more basic type formers, such as the empty type and coproducts. In the resulting theory we can define equivalent types that possess all expected judgmental equalities. Our overall result is, that we can start with a theory that contains dependent functions and pairs, natural numbers, paths, univalent universes and propositional resizing and obtain all strictly positive types. Most of our work is formalized in Coq. There are some unformalized parts however, because Coq does not support propositional resizing.

1.1 Related Work

Abbott et al. [1] described the category of containers in a type theory with uniqueness of identity proofs and showed that W-types are sufficient to construct all strictly positive types in such a type theory. Altenkirch et al. [3] extended this work to indexed containers that capture also inductive and coinductive families of types. Ahrens et al. [2] translated the construction of M-types from natural numbers to the proof relevant setting of HoTT.

1.2 Contributions

- Construction of strictly positive types in HoTT
- W-types from natural numbers
- M-types with judgmental computation rule

2 Background

In this chapter we introduce some basic concepts and notations that we will use throughout the thesis. The notations are the same as in the HoTT book [4]. In the first section we describe the core of the type theory that we will use. The second section contains type formers that we will assume for the moment. In later chapters we will see however that those type formers can be replaced by definitions in the core theory. Other sections of this chapter present some basic definitions and lemmas from HoTT and add two important extensions to our type theory which are *univalence* and *propositional resizing*. To the end of this chapter we will introduce *containers* and a characterization of inductive and coinductive types which is central to our work.

2.1 The Core Type Theory

We use a subset of the type theory from the HoTT book [4] which is an extension of Martin-Löf type theory. We only give a short, informal overview here. For a more extensive presentation we refer to the HoTT book. We think of types as set-like objects. There are two judgments that allow us to make statements about types and their elements:

(i) The judgment $x : A$ expresses that x is an *element* of the type A. We also say, x *has type* A or x *inhabits* A.
(ii) The judgment $x \equiv y$ expresses an equality between x and y. We read this as, x is *judgmentally equal* to y or x is equal to y *by definition.*

The notation $x :\equiv y$ indicates that we define x to be equal to y. The type theory consists of a set of inference rules that allow us to conclude judgments. There are rules that make judgmental equality an equivalence relation and a rule that makes judgmentally equal types interchangable i.e. if x has type A and A is judgmentally equal to B then x has type B. Other rules can be grouped into type formers which are sets of rules that describe how a certain type behaves. This usually means that they allow us to build and destruct elements of a type and give us judgmental equalities for those operations, which we call computation rules.

2.1.1 Functions

Fix a type A, and a term Φ that describes a type with free variable x. The term $\prod_{(x:A)} \Phi$ stands for the type of dependent functions that take an argument $a : A$ and return an element of $\Phi[a/x]$ which stands for Φ with every free occurence of x replaced by a. Often we ommit the type annotation for binders, which means in the case of function types that we just write $\prod_{(x)} \Phi$. Sometimes we use a wildcard (_) as in $\prod_{(\_:A)} \Phi$ to indicate that we won't use the argument and don't want to assign a name to it. If Φ does not contain x as free variable, we also write $A \to B$ for the type of functions from A to B.

If $f : \prod_{(x:A)} \Phi$ is a function and $a : A$ then we can apply f to a. The result $f(a)$ has type $\Phi[a/x]$. We build functions as abstractions: Given a term $\phi : \Phi$ with free variable x, the abstraction $\lambda(x : A).\, \phi$ has type $\prod_{(x:A)} \Phi$. As a shorthand for abstraction we use the blank symbol $(-)$ to indicate a bound variable. For example the term $f(-)$ stands for the abstraction $\lambda x.\, f(x)$. If we want to assign a name to a function, we also use the syntax $f(x) :\equiv \Phi$ to define $f : \prod_{(x:A)} \Phi$.

There are two computation rules for functions. The first describes the application of an abstraction: $(\lambda x.\, \Phi)(a) \equiv \Phi[a/x]$. The second computation rule enables us to unfold trivial abstractions: $\lambda x.\, f(x) :\equiv f$ for all functions f.

We define functions with multiple arguments by *currying*, which means for example that a function f that takes arguments $x : A$ and $y : B(x)$ and returns an element of $C(x, y)$ has type $\prod_{(x:A)} \prod_{(y:B(x))} C(x, y)$. If we give both arguments at the same time, we usually write this as $f(a, b)$.

2.1.2 Pairs

Fix a type A, and a term Φ that describes a type with free variable x. The term $\sum_{(x:A)} \Phi$ stands for the type of dependent pairs that consist of a component $a : A$ and a component $b : \Phi[a/x]$. If Φ has no free occurence of x, we use the notation $A \times \Phi$. We construct a pair as (a, b). Given a pair $p : \sum_{(x:A)} \Phi$, we denote both components by $\mathsf{pr}_1(p) : A$ and $\mathsf{pr}_2(p) : \Phi[\mathsf{pr}_1(p)/x]$ respectively. We have two computation rules that describe the result of projections on a constructor application: $\mathsf{pr}_1((x, y)) \equiv x$ and $\mathsf{pr}_2((x, y)) \equiv y$. A third computation rule describes the application of the constructor on projections: $(\mathsf{pr}_1(p), \mathsf{pr}_2(p)) \equiv p$. Sometimes we use the pair notation in binders as in $\lambda(x, y).\, \phi$ or $\prod_{((x,y))} \Phi$, which means that x and y stand for the components of the pair.

2.1.3 Natural Numbers

We denote by $\mathbb{N}$ the type of natural numbers. There are two constructors $0 : \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. We also write 1 for $\mathsf{succ}(0)$, 2 for $\mathsf{succ}(\mathsf{succ}(0))$ etc. and also $n + 1$ for $\mathsf{succ}(n)$, $n + 2$ for $\mathsf{succ}(\mathsf{succ}(n))$ .... For natural numbers we have an induction principle

$$\mathsf{ind}_{\mathbb{N}} : \prod_{P:\mathbb{N}\to\mathcal{U}} P(0) \to \left(\prod_{n:\mathbb{N}} P(n) \to P(\mathsf{succ}(n))\right) \to \prod_{n:\mathbb{N}} P(n)$$

with computation rules

$$\begin{aligned} \mathsf{ind}_{\mathbb{N}}(P, s, f, 0) &\equiv s \\ \mathsf{ind}_{\mathbb{N}}(P, s, f, \mathsf{succ}(n)) &\equiv f(\mathsf{ind}_{\mathbb{N}}(P, s, f, n)). \end{aligned}$$

We use a pattern-matching notation for the definition of recursive functions on natural numbers and later also for other types. The definition

$$\begin{aligned} r &: \textstyle\prod_{(n)} P(n) \\ r(0) &:\equiv s \\ r(\mathsf{succ}(n)) &:\equiv f(r(n)) \end{aligned}$$

stands for $r :\equiv \mathsf{ind}_{\mathbb{N}}(P, s, f)$.

2.1.4 Paths

Given two objects $x, y : A$, there is a type $x = y$ of *paths* from x to y, which we also call *propositional equalities*. If the type $x = y$ is inhabited, we say that x and y are propositionally equal or just that x is equal to y. There is a single constructor $\mathsf{refl}_{(-)} : \prod_{(x:A)} x = x$. The induction principle, which we call *path induction*, states that if have $x, y : A$ and a path $p : x = y$, we can assume that y is x and p is refl_x:

$$\mathsf{ind}_{=_A} : \prod_{P : \prod_{(x,y:A)} (x=y) \to \mathcal{U}} \left(\prod_{x} P(x, x, \mathsf{refl}_x) \right) \to \prod_{(x,y:A)} \prod_{(p:x=y)} P(x, y, p).$$

The computation rule is

$$\mathsf{ind}_{=_A}(P, f, x, x, \mathsf{refl}_x) \equiv f(x).$$

Our treatment of equality as the type of paths is different from that which was used for the construction of strictly positive types by Abbott et al. [1]. There might be different proofs of equality between two objects. In fact the type of paths between objects can have very interesting structures It is the core of HoTT reason about these structures.

2.1.5 Universes

Universes are types that contain types. We have an infinite hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \cdots$$

This hierarchy is cumulative which means that every element of one universe is also contained in all bigger universes. All types are contained in some universe. The universe of a type is generally the biggest of all universes of types that were used in its construction. For example assume that $A : \mathcal{U}_m$ and $B : \mathcal{U}_n$. Then $A \to B$ has universe level $\max(m, n)$. A consequence is that the type $A :\equiv \mathcal{U}_n \to B$ has universe level at least $n + 1$ and thus a function $f : A$ cannot be applied to A. We will usually leave the universe index implicit. Where it is important, we will mention it explicitly.

2.2 Admissible Type Formers

Now we list some type formers that we will define from elements in our core theory later on. All of our redefinitions will have the same properties as introduced here except for W-types which will have less judgmental computation rules.

2.2.1 Empty Type

We denote by $\mathbf{0}$ the empty type which has no constructor. The induction principle states that we can construct everything from an element of $\mathbf{0}$.

$$\mathsf{ind}_{\mathbf{0}} : \prod_{C} c.$$

2.2.2 Unit

We denote by $\mathbf{1}$ the type with exactly one element. It has one constructor $\star : \mathbf{1}$ and an induction principle

$$\mathsf{ind}_{\mathbf{1}} : \prod_{C:\mathbf{1}\to\mathcal{U}} C(\star) \to \prod_{x:\mathbf{1}} C(x).$$

We also have a computation rule

$$\mathsf{ind}_{\mathbf{1}}(C, c, \star) \equiv c.$$

2.2.3 Bool

We denote by $\mathbf{2}$ the boolean type with exactly two elements. It has constructors $1_{\mathbf{2}} : \mathbf{2}$ and $0_{\mathbf{2}} : \mathbf{2}$ and an induction principle

$$\mathsf{ind}_{\mathbf{2}} : \prod_{C:\mathbf{2}\to\mathcal{U}} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \prod_{x:\mathbf{2}} C(x)$$

with computation rules

$$\mathsf{ind}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) \equiv c_0$$
$$\mathsf{ind}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) \equiv c_1.$$

2.2.4 Coproducts

Given types A and B, the coproduct $A+B$ is the type theoretic equivalent to the disjoint union in set theory. Every element of $A+B$ is either an element of A or an element of B. We have two constructors $\mathsf{inl} : A \to A+B$ and $\mathsf{inr} : B \to A+B$. The induction principle has type

$$\mathsf{ind}_{A+B} : \sum_{C:A+B\to\mathcal{U}} \left(\prod_{a:A} C(\mathsf{inl}(a))\right) \to \left(\prod_{b:B} C(\mathsf{inr}(b))\right) \to \prod_{x:A+B} C(x).$$

There are two computation rules

$$\mathsf{ind}_{A+B}(C, g_0, g_1, \mathsf{inl}(x)) \equiv g_0(x)$$
$$\mathsf{ind}_{A+B}(C, g_0, g_1, \mathsf{inr}(x)) \equiv g_1(x).$$

2.2.5 Options

Given a type A, the option type $\mathsf{Option}(A)$ extends A by one element. There are two constructors $\mathsf{none} : \mathsf{Option}(A)$ and $\mathsf{some} : A \to \mathsf{Option}(A)$. We have an induction principle

$$\mathsf{ind}_{\mathsf{Option}(A)} : \prod_{C:\mathsf{Option}(A)\to\mathcal{U}} \left(\prod_{a:A} C(\mathsf{some}(a))\right) \to C(\mathsf{none}) \to \prod_{x:\mathsf{Option}(A)} C(x)$$

and two computation rules

$$\mathsf{ind}_{\mathsf{Option}(A)}(C, c_{\mathsf{some}}, c_{\mathsf{none}}, \mathsf{some}(x)) \equiv c_{\mathsf{some}}(x)$$
$$\mathsf{ind}_{\mathsf{Option}(A)}(C, c_{\mathsf{some}}, c_{\mathsf{none}}, \mathsf{none}) \equiv c_{\mathsf{none}}.$$

2.2.6 W-Types

Given a type A and a family $B : A \to \mathcal{U}$, the elements of the W-type $\mathsf{W}_{(a:A)}B(a)$ represent well-founded trees with nodes labeled by elements of A. If a node is labeled with some $a : A$, the type $B(a)$ gives us the *number* of subtrees at that node. The type is generated by one constructor $\mathsf{sup} : \prod_{(a:A)}(B(a) \to \mathsf{W}_{(a:A)}B(a)) \to \mathsf{W}_{(a:A)}B(a)$. We have an induction principle

$$\mathsf{ind}_{\mathsf{W}_{(a:A)}B(a)} : \prod_{(C:\mathsf{W}_{(a:A)}B(a)\to\mathcal{U})} \left(\prod_{(a:A)} \prod_{(f:B(a)\to\mathsf{W}_{(a:A)}B(a))} \left(\prod_{b:B(a)} C(f(b))\right) \to C(\mathsf{sup}(a,f))\right) \to \prod_{x:\mathsf{W}_{(a:A)}B(a)} C(x).$$

This satisfies the computation rule

$$\mathsf{ind}_{\mathsf{W}_{(a:A)}B(a)}(C, g, \mathsf{sup}(a,f)) \equiv g(a, f, \mathsf{ind}_{\mathsf{W}_{(a:A)}B(a)}(C,g) \circ f).$$

As for all other types we will use the more readable pattern-matching notation for the definition of rekursive functions on a W-type.

2.3 Propositions as Types

We will use types to express propositions, as it is common practice in type theory. Every element of the type is considered a proof for this proposition.

- The type $A \to B$ expresses that A implies B.
- The type $\prod_{(x:A)} B(x)$ expresses that all $x : A$ satisfy B.
- The type $A \times B$ expresses that both A and B are true.
- The type $\sum_{(x:A)} B(x)$ expresses that there is some $x : A$ that satisfies B.
- The type $x = y$ expresses that x is equal to y.

2.4 Working with Paths

We will now introduce some operations that allow us to reason about equality. The proofs are all by path induction.

Fact 2.4.1. *Given paths $p : x = y$ and $q : y = z$, there is a path $p \cdot q : x = z$.*

Fact 2.4.2. *Given elements a path $p : x = y$, there is a path $p^{-1} : y = x$.*

Lemma 2.4.3. *If we apply a function to equal terms, we get equal results: For all functions $f : A \to B$, elements $x, y : A$ and paths $p : x = y$ we have a path*

$$\mathsf{ap}_f(p) : f(x) = f(y).$$

Proof. By path induction we only need to consider the case where y is x and p is the reflexivity path. In this case we define $\mathsf{ap}_f(\mathsf{refl}_x) :\equiv \mathsf{refl}_{f(x)}$. □

If we have types A and B, an element $x : A$ and a propositional equality $p : A = B$, one could expect that we also have $x : B$, but in general this is not the case. We can however apply the following slightly more general lemma to obtain an element of B in such a setting.

Lemma 2.4.4. *Given a family of types $B : A \to \mathcal{U}$, an equality $p : x = y$ between elements of A and an element $b : B(x)$, there is an inhabitant*

$$\mathsf{transport}^B(p, y) : B(y).$$

If the family B is clear from the context, we use the shorter notation $p_*(b) : B(y)$.

Proof. By path induction we only need to consider the case where y is x and p is the reflexivity path, hence $B(x) \equiv B(y)$. For this case we define $\mathsf{transport}^B(\mathsf{refl}_x, b) :\equiv b$. □

The application of this lemma is also called rewriting because it allows us to *rewrite* parts of types, especially in propositions that we want to prove.

When working with transport, we often need to know what it does on certain types. The following equalities can often be used to compute the result of a transport.

Fact 2.4.5.

$$\mathsf{transport}^{\lambda x.\, A(x) \to B(x)}(p, f)(a) = \mathsf{transport}^B(p, f(\mathsf{transport}^A(p, a))).$$

Fact 2.4.6.

$$\mathsf{transport}^{\lambda x.\, A(x) \times B(x)}(p, (a, b)) = (\mathsf{transport}^A(p, a), \mathsf{transport}^B(p, b)).$$

2.5 Equivalence

Types that are defined differently can still behave very similar. For example the products $A \times B$ and $B \times A$ allow us to do exactly the same things. This also holds for the types A and $A + \mathbf{0}$ and many others. We express this similarity with the notion of *equivalence*. For its definition we need to talk about inverses of a function.

Definition 2.5.1. Given a function $f : A \to B$, another function $g : B \to A$ is a **left inverse** to f if for all $x : A$ we have $g(f(x)) = x$.

Definition 2.5.2. Given a function $f : A \to B$, another function $g : B \to A$ is a **right inverse** to f if for all $x : B$ we have $f(g(x)) = x$.

Definition 2.5.3. A function $f : A \to B$ is an **equivalence** if there is a left inverse and a right inverse.

There is a reason not to demand that the left inverse and right inverse are the same, but it is not relevant for our work. For more information we refer to chapter 4 [4]. When we explicitly construct an equivalence, we will always use the same function for both. In that case we call it a pseudo-inverse.

Definition 2.5.4. Given a function $f : A \to B$, another function $g : B \to A$ is a **pseudo-inverse** to f if it is a left inverse and a right inverse.

Definition 2.5.5. Two types A and B are equivalent ($A \simeq B$) if there is an equivalence from A to B.

We will see that type equivalence is really an equivalence relation. We show this step by step because in general there can be many equivalences between two types. We need to compute with equivalences, thus it matters how exactly they are defined.

Fact 2.5.6. *The identity function* id_A *is an equivalence. Hence type equivalence is reflexive.*

Fact 2.5.7. *If* $f : A \to B$ *is an equivalence and* $g : B \to C$ *is an equivalence, the composition* $g \circ f : A \to C$ *is also an equivalence. Hence type equivalence is transitive.*

Fact 2.5.8. *Every equivalence* f *has a unique pseudo-inverse. We denote it by* f^{-1}. *Hence type equivalence is symmetric.*

Corollary 2.5.9. *Type equivalence is an equivalence relation.*

2.6 Univalence

Now we will proceed to introduce the univalence axiom which is likely the most important innovation from HoTT. It is based on the observation that there is no way to distinguish equivalent types inside the type theory. Moreover we can use an equivalence to manually *transport* definitions that were done on one type to the other type. Thus it seems natural to assume that equivalent types are equal. This is a consequence of the univalence axiom. But the univalence axiom is even stronger. It doesn't just state the equivalence implies equality, it asserts that equivalence is equivalent to equality. This means that there are exactly as many paths between two types as there are equivalences. We can define one direction of the equivalence without further axioms:

Lemma 2.6.1. *For every equality $p : A = B$ between types, there is an equivalence*

$$\mathsf{idtoeqv}(p) : A \simeq B.$$

Proof. By path induction and reflexivity . □

Axiom 2.6.2 (Univalence)**.** *For all types A and B the function $\mathsf{idtoeqv} : (A = B) \to (A \simeq B)$ is an equivalence.*

Definition 2.6.3. For every equivalence $f : A \to B$ we define

$$\mathsf{ua}(f) : A = B$$

as the unique pseudo inverse of $\mathsf{idtoeqv}$.

Sometimes we transport over a path that was built by univalence. In such cases the transport does the same as an application of the underlying equivalence:

Fact 2.6.4. *For all equivalences $e : A \to B$ and inhabitants $x : A$ we have*

$$\mathsf{transport}^{\mathsf{id}}(\mathsf{ua}(e), x) = e(x).$$

One welcome side effect of univalence is function extensionality. Similar to the definition of univalence this does not only mean that pointwise equality on functions implies equality. It means that for all functions f and g the following is an equivalence.

Lemma 2.6.5. *For all functions $f, g : \prod_{(x:A)} \to B(x)$ and paths $p : f = g$ there is a pointwise equality*

$$\mathsf{happly}(p) : \left(\prod_x f(x) = g(x)\right).$$

Proof. By path induction. □

Fact 2.6.6. *For $f, g : \prod_{(x:A)} \to B(x)$ the function $\mathsf{happly} : (f = g) \to \left(\prod_{(x)} f(x) = g(x)\right)$ is an equivalence.*

Definition 2.6.7. For all functions $f, g : \prod_{(x:A)} \to B(x)$, we define

$$\mathsf{funext} : \left(\prod_x f(x) = g(x)\right) \to (f = g)$$

as the unique pseudo-inverse of happly.

2.7 Mere Propositions

Definition 2.7.1. A type is a **mere proposition** if it has at most one inhabitant:

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} x = y.$$

We define the type of mere propositions as

$$\mathsf{Prop} :\equiv \sum_{A:\mathcal{U}} \mathsf{isProp}(A)$$

but identify inhabitants of Prop with their first component. If the codomain of a function is a mere proposition, we call it a mere predicate.

We call such a type A a mere propositions because an element doesn't give us any more information than the fact that A holds as a proposition. Hence any other interpretation of A but that as a proposition is not very interesting in most cases. However we will see that mere propositions can also be useful through the judgmental equalities on their elements. To exploit this we will need **propositional resizing** as an extension to our type theory.

Axiom 2.7.2 (Propositional Resizing)**.** *Every mere proposition inhabits the smallest universe.*

For the moment we assume a type former that is related to mere propositions. Given $A : \mathcal{U}$, the type $\|A\|$ is **propositional truncation** of A. The propositional truncation is a mere proposition i.e. every two inhabitants are equal. It has one constructor $|{-}| : A \to \|A\|$ and an elimination principle

$$\mathsf{rec}_{\|A\|} : \prod_{P:\mathsf{Prop}} (A \to P) \to \|A\| \to P.$$

The computation rule gives us an equality $\mathsf{rec}_{\|A\|}(P, f, |a|) \equiv f(a)$ for all f and a.

2.8 Contractibility

Sometimes we say that a type has exactly one inhabitant. We will now give a formal definition for this statement in the form of contractibility.

Definition 2.8.1. A type is **contractible** if it has exactly one inhabitant:

$$\mathsf{isContr}(A) :\equiv \sum_{(x:A)} \prod_{(y:A)} x = y.$$

Every contractible type is also a mere proposition:

Fact 2.8.2. *Given two inhabitants x and y of a contracible type A, there is a path* $\mathsf{path\_contr}(x, y) : x = y$.

There is one form of contractible types that we will use throughout this thesis.

Fact 2.8.3. *For all types A and inhabitants $a : A$ the type $\sum_{(x:A)} a = x$ is contractible.*

When we prove contractibility of function-types or products, we can use the following facts.

Fact 2.8.4. *The type $\prod_{(x:A)} B(x)$ is contractible if for all $x : A$ the type $B(x)$ is contractible.*

Fact 2.8.5. *The type $\sum_{(x:A)} B(x)$ is contractible if A is contractible and for all $x : A$ the type $B(x)$ is contractible.*

Fact 2.8.6. *Given a type A, the type* $\mathsf{isContr}(A)$ *is contractible.*

2.9 Inductive Types

In this chapter we introduce a characterization of inductive types. We could characterize inductive types as types with certain constructors and an induction principle, but we choose another way because we want it to be dual for coinductive types. For our characterization we need the notion of a functor.

Definition 2.9.1. Given a type I, a functor F consists of

- A function $F_0 : \mathcal{U} \to \mathcal{U}$
- For all types A and B a function $F_{A,B} : (A \to B) \to F_0(A) \to F_0(B)$
- For all types A a proof $F_{A,A}\mathsf{id} = \mathsf{id}$
- For all types A, B and C and functions $f : A \to B$ and $g : B \to C$ a proof $F_{A,C}(g \circ f) = F_{B,C}(g) \circ F_{A,B}(f)$

We adopt the often seen notation from category theory and write $F\,A$ for $F_0(A)$, if A is a type and $F\,f$ for $F_{A,B}(f)$, if f is a function from A to B.

Intuitively we can think of functors as abstract data structures. A value of type $F\,A$ may contain elements of A and $F_{A,B}$ allows us to replace those by elements of B. The additional rules make sure that $F_{A,B}$ really replaces components and doesn't permute them on the way. When we define a functor, we often give only F_0 and $F_{A,B}$ and sometimes not even $F_{A,B}$. The definition of the remaining components is routine and not interesting for us.

Now let us take a look at things that we expect from an inductive type. First of all, an inductive type should have constructors. For example the type of natural numbers has constructors $0 : \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. But we could as well define it with a single constructor $s_{\mathbb{N}} : \mathbf{1} + \mathbb{N} \to \mathbb{N}$. In which case the *signature* of the constructor is given by the functor $F_{\mathbb{N}}\,X :\equiv \mathbf{1} + X$. We generalize this concept of types with a constructor for a functorial signature as algebras.

Definition 2.9.2. Given a functor F, an **F-algebra** is a type A with an algebra structure $s_A : F\,A \to A$.

Another example is the type $\mathbf{T_2}$ of binary trees, which can be defined with a constructor $s_{\mathbf{T_2}} : \mathbf{1} + (\mathbf{T_2} \times \mathbf{T_2}) \to \mathbf{T_2}$ and thus gives rise to an F-algebra for the functor defined by $F_{\mathbf{T_2}}\,X :\equiv \mathbf{1} + (X \times X)$.

But an inductive type should not only have a constructor. We also expect a way to define functions on it by recursion. On natural numbers we can use any start value $0_X : X$ and step function $\mathsf{succ}_X : X \to X$ to uniquely define a function $r : \mathbb{N} \to X$ which iterates succ_X and satisfies the computation rules

$$\begin{aligned} r(0) &\equiv 0_X \\ r(\mathsf{succ}(n)) &\equiv \mathsf{succ}_X(r(n)). \end{aligned}$$

To generalize this, note that X is a $F_{\mathbb{N}}$-algebra with algebra structure

$$\begin{aligned} s_X(\mathsf{inl}(\star)) &:\equiv 0_X \\ s_X(\mathsf{inr}(x)) &:\equiv \mathsf{succ}_X(x). \end{aligned}$$

With this structure we can write the computation rule as

$$\prod_{x:F\,\mathbb{N}} r(s_{\mathbb{N}}(x)) = s_X(F(r)(x)).$$

We generalize the type of such functions with computation rule by the definition of algebra morphisms.

Definition 2.9.3. Given F-algebras (A, s_A) and (B, s_B), an **algebra morphism** from (A, s_A) to (B, s_B) consists of a function from A to B and a proof of the computation rule:

$$\mathsf{AlgHom}((A, s_A), (B, s_B)) :\equiv \sum_{(h:A\to B)} \prod_{(x:F\,A)} h(s_A(x)) = s_B(F(h)(x)).$$

The algebra morphisms from $(\mathbb{N}, s_{\mathbb{N}})$ correspond to iterative functions. Now that we can describe types with constructors as algebras and iterative functions as algebra morphisms, we can describe inductive types as algebras that have exactly one morphism to every other algebra.

Definition 2.9.4. An **initial F-algebra** is an F-algebra A such that for every second F-algebra B the type of algebra morphisms from A to B is contractible.

It is clear that every inductive type is an initial algebra but the following fact shows that the definition of initial algebras is already strict enough to exclude everything else.

Fact 2.9.5. *There is at most one initial algebra for every functor.*

This still doesn't mean that we can do everything with initial algebras, that we expect from inductive types. But we will see that every initial algebra has an induction principle that satisfies at least a propositional computation rule. The induction principle on natural numbers has type

$$\begin{aligned}
&\prod_{P:\mathbb{N}\to\mathcal{U}} P(0) \to \left(\prod_{n} P(n) \to P(n+1)\right) \to \prod_{n} P(n) \\
&\simeq \prod_{P:\mathbb{N}\to\mathcal{U}} P(0) \to \left(\prod_{(n,p):\sum_{(n)} P(n)} P(n+1)\right) \to \prod_{n} P(n) \\
&\simeq \prod_{x:F_{\mathbb{N}}(\sum_{(n)} P(n))} P(s_{\mathbb{N}}(F_{\mathbb{N}}(\mathsf{pr}_1)(x))) \to \prod_{n} P(n).
\end{aligned}$$

The induction scheme for every other algebra can be expressed in the same form, which makes it possible to formulate a general induction principle for initial algebras.

Fact 2.9.6. *Every initial F-algebra (A, s_A) has an induction induction principle*

$$\mathsf{ind}_{(A,s_A)} : \prod_{x:F(\sum_{(a)} P(a))} P(s_A(F(\mathsf{pr}_1)(x))) \to \prod_{a} P(a).$$

It satisfies the computation rule

$$\mathsf{ind}_{(A,s_A)}(f, s_A(x)) = f(F(\lambda a.\,(a, \mathsf{ind}_{(A,s_A)}(f, a)))(x))$$

for all f and x.

2.10 Coinductive Types

Coinductive types have a destructor and a unique corecursion function that constructs elements of the type from a step function. This is completely dual to the definition of initial algebras. We generalize types with a destructor as coalgebras.

Definition 2.10.1. Given a functor F, an **F-coalgebra** is a type A with a coalgebra structure $s_A : A \to F\,A$.

The computation rule for corecursion gives us the result of destructor applications on elements that were corecursively constructed. We generalize corecursive functions as coalgebra morphisms.

Definition 2.10.2.

$$\mathsf{CoalgHom}((A, s_A), (B, s_B)) :\equiv \sum_{(h:A\to B)} \prod_{(a)} s_B(h(a)) = F(h)(s_A(a)).$$

Finally we generalize coinductive types as final coalgebras.

Definition 2.10.3. A **final F-coalgebra** is a coalgebra such that the type of morphisms to every other algebra is contractible.

Here we also have a uniqueness result.

Fact 2.10.4. *For every functor F there is at most one final F-algebra.*

There is no dual concept to the induction principle. There is coinduction which can be used to prove equality between elements of a coinductive type from bisimulation, but we will not use that for our work.

2.11 Addresses

W-types describe tree-like structures. We want to define a type of addresses in such structures, but we want to use it with coinductive types as well. Hence we generalize. Fix types T and A and a family $B : A \to \mathcal{U}$ and assume that we have two destructors $\mathsf{label} : T \to A$ and $\mathsf{arg} : \prod_{(t:T)} B(\mathsf{label}(t)) \to T$. We think of an element $t : T$ as a tree with label $\mathsf{label}(t)$ at the root and subtrees $\mathsf{arg}(w, b)$ for $b : B(\mathsf{label}(t))$. We want the type $\mathsf{Addr}(t)$ of addresses to be the indexed inductive type with two constructors

$$\begin{aligned}
\mathsf{root\_addr} &: \prod_{w:\mathsf{W}} \mathsf{Addr}(w) \\
\mathsf{subtree\_addr} &: \prod_{(w:\mathsf{W})} \prod_{(b:B(\mathsf{label}(w)))} \prod_{(addr':\mathsf{Addr}(\mathsf{arg}(w,b)))} \mathsf{Addr}(w).
\end{aligned}$$

Since we want to keep our type theory small, we construct this type from the core theory.

Definition 2.11.1. We define the types of addresses with fixed depth by recursion and then define $\mathsf{Addr}(t)$ as disjoint union of those types:

$$\begin{aligned}
\mathsf{Addr}_0(t, 0) &:\equiv \mathbf{1} \\
\mathsf{Addr}_0(t, \mathsf{succ}(n)) &:\equiv \sum_{b:B(\mathsf{label}(t))} \mathsf{Addr}_0(\mathsf{arg}(t, b), n) \\
\mathsf{Addr}(t) &:\equiv \sum_{n:\mathbb{N}} \mathsf{Addr}'(t, n).
\end{aligned}$$

Definition 2.11.2. We define the constructors

$$\begin{aligned}
\mathsf{root\_addr} &:\equiv (0, \star) \\
\mathsf{subtree\_addr}(b, (n, x)) &:\equiv (\mathsf{succ}(n), (b, x)).
\end{aligned}$$

Lemma 2.11.3. *There is an induction principle of type*

$$\mathsf{ind}_{\mathsf{Addr}} : \prod_{P:\prod_{(w:\mathsf{W})}\mathsf{Addr}(w)\to\mathcal{U}} \left(\prod_{w:\mathsf{W}} P(w,\mathsf{root\_addr})\right) \to$$
$$\left(\prod_{(w)}\prod_{(b)}\prod_{(addr:\mathsf{Addr}(\mathsf{arg}(w,b)))} P(\mathsf{arg}(w,b), addr) \to P(w, \mathsf{subtree\_addr}(b, addr))\right) \to$$
$$\prod_{(w:\mathsf{W})}\prod_{(addr:\mathsf{Addr}(w))} P(w, addr).$$

Proof. By recursion on the depth of the address. □

We will use the familiar pattern-matching notation on addresses as syntactic sugar. The following remark asserts that the resulting functions satisfy the expected computation rules by definition.

Remark 2.11.4. The computation rules for $\mathsf{ind}_{\mathsf{Addr}}$ hold judgmentally.

Now we define three functions on addresses that we will use later.

Definition 2.11.5. We define a function $\mathsf{subtree\_at} : \prod_{(w:\mathsf{W})}\mathsf{Addr}(w) \to \mathsf{W}$, that returns the subtrees at given addresses by recursion on the address and a function $\mathsf{label\_at} : \prod_{(w:\mathsf{W})}\mathsf{Addr}(w) \to A$, that returns the label at an address:

$$\begin{aligned}\mathsf{subtree\_at}(w,\mathsf{root\_addr}) &:\equiv w\\ \mathsf{subtree\_at}(w,\mathsf{subtree\_addr}(b, addr)) &:\equiv \mathsf{subtree\_at}(\mathsf{arg}(w,b), addr)\\ \mathsf{label\_at}(w, addr) &:\equiv \mathsf{label}(\mathsf{subtree\_at}(w, addr)).\end{aligned}$$

Definition 2.11.6. We define a function

$$\mathsf{extend\_addr} : \prod_{(w:\mathsf{W})}\prod_{(addr:\mathsf{Addr}(w))} B(\mathsf{label\_at}(w, addr)) \to \mathsf{Addr}(w)$$

that extends an address to an address at a deeper level:

$$\begin{aligned}\mathsf{extend\_addr}(w,\mathsf{root\_addr}, b) &:\equiv \mathsf{subtree\_addr}(b,\mathsf{root\_addr})\\ \mathsf{extend\_addr}(w,\mathsf{subtree\_addr}(b', addr), b) &:\equiv \mathsf{subtree\_addr}(b',\\ &\qquad \mathsf{extend\_addr}(\mathsf{arg}(w,b'), addr, b)).\end{aligned}$$

We also use a slightly different familly of types $\mathsf{Addr}' : A \to T \to \mathcal{U}$. The inhabitants of such a type $\mathsf{Addr}'(a,t)$ represent addresses that point to node that is labeled with a. The definition is analogous to that of Addr.

Fact 2.11.7. *There is a family of types* $\mathsf{Addr}' : A \to T \to \mathcal{U}$ *with constructors*

$$\begin{aligned}\mathsf{root\_addr}' &: \prod_{w:\mathsf{W}} \mathsf{Addr}'(\mathsf{label}(w), w)\\ \mathsf{subtree\_addr}' &: \prod_{(w:\mathsf{W})} \prod_{(b:B(\mathsf{label}(w)))} \prod_{(addr':\mathsf{Addr}'(a,\mathsf{arg}(w,b)))} \mathsf{Addr}(a, w).\end{aligned}$$

and an induction principle

$$\begin{aligned}&\mathsf{ind}_{\mathsf{Addr}'} : \prod_{P:\prod_{(a:A)}\prod_{(w:\mathsf{W})} \mathsf{Addr}'(a,w)\to\mathcal{U}} \left(\prod_{w:\mathsf{W}} P(\mathsf{label}(w), w, \mathsf{root\_addr}')\right) \to\\ &\left(\prod_{(a,w,b)} \prod_{(addr:\mathsf{Addr}'(a,\mathsf{arg}(w,b)))} P(a, \mathsf{arg}(w,b), addr) \to P(a, w, \mathsf{subtree\_addr}'(b, addr))\right) \to\\ &\qquad \prod_{(a:A)} \prod_{(w:\mathsf{W})} \prod_{(addr:\mathsf{Addr}'(a,w))} P(a, w, addr).\end{aligned}$$

This type also satisfies the computation rule judgmentally.

2.12 Containers

We use containers as normal form that can represent many types with free variables. The most primitive form of a container represents types in a single free variable and describes a functor.

Definition 2.12.1. A **container** $S \blacktriangleright P$ consists of a type of shapes S and a family $P : S \to \mathcal{U}$.

Definition 2.12.2. The **extension of a container** $S \blacktriangleright P$ is the functor $(\!|S \blacktriangleright P|\!)$ with

$$\begin{aligned}&(\!|S \blacktriangleright P|\!)\,\Gamma : \mathcal{U}\\ &(\!|S \blacktriangleright P|\!)\,\Gamma :\equiv \textstyle\sum_{(s:S)} P(s) \to \Gamma\\[4pt] &(\!|S \blacktriangleright P|\!)\,f : (\!|S \blacktriangleright P|\!)\,\Gamma \to (\!|S \blacktriangleright P|\!)\,\Gamma'\\ &(\!|S \blacktriangleright P|\!)\,f :\equiv \lambda(s, g).\,(s, f \circ g)\end{aligned}$$

for all $\Gamma, \Gamma' : \mathcal{U}$ and $f : \Gamma \to \Gamma'$.

We have already seen the type of lists as an example for this. Container functors are interesting because they have an initial algebra and a final coalgebra.

Fact 2.12.3. *For all types* A *and families* $B : A \to \mathcal{U}$*, the type* $\mathsf{W}_{(a:A)}B$ *is an initial algebra of the functor* $(\!|A \blacktriangleright B|\!)$*.*

We will see in the next chapter how to construct a final coalgebra for container functors. Now we define a more general form of containers that represent types with multiple free variables.

Definition 2.12.4. An **I-indexed container** $S \triangleright P$ consists of a type of shapes S and a family $P : I \to S \to \mathcal{U}$.

The index I represents the type of free variables.

Definition 2.12.5. The **extension of an I-indexed container** $S \triangleright P$ is the function $[\![S \triangleright P]\!]$ with

$$\begin{aligned} &[\![S \triangleright P]\!]\,\Gamma : \mathcal{U} \\ &[\![S \triangleright P]\!]\,\Gamma :\equiv \textstyle\sum_{(s:S)}\prod_{(i)} P(i,s) \to \Gamma(i) \end{aligned}$$

for all $\Gamma : I \to \mathcal{U}$. For applications of this function we use the same notation as for functors although it is not a functor by our formal definition.

A simple example is the $\mathbf{2}$-indexed container

$$c :\equiv \mathbf{1} \triangleright \lambda i.\, \lambda s.\, \mathbf{1}.$$

For all environments $\Gamma : \mathbf{2} \to \mathcal{U}$ we have

$$[\![c]\!]\,\Gamma \equiv \sum_{(\_:\mathbf{1})} \prod_{(i:\mathbf{2})} \mathbf{1} \to \Gamma(i)$$

which is equivalent to the product $\Gamma(0_{\mathbf{2}}) \times \Gamma(1_{\mathbf{2}})$.

3 Construction of M-Types

In this chapter we present a construction of M-types which are final coalgebras for container functors. Analogously to W-types we think of M-types as trees, with the difference that an element of a M-type does not need to be well-founded. For the remainder of the chapter fix a type A and a family $B : A \to \mathcal{U}$. We build on a construction of final coalgebras for container functors by Ahrens et al. [2] and refine it to obtain the computation rule as judgmental equality.

Fact 3.0.1. *There is a final coalgebra* $(\mathsf{M}_0, s_{\mathsf{M}_0})$ *for the container functor* $[\![A \rhd B]\!]$.

We denote the corecursor on this coalgebra by $\mathsf{corec}_{\mathsf{M}_0}$. We use $(\mathsf{M}_0, s_{\mathsf{M}_0})$ to define a new coalgebra $(\mathsf{M}, s_{\mathsf{M}})$ and explicitly give a corecursor that satisfies the computation rule judgmentally. Finally we will prove the new coalgebra equal to $(\mathsf{M}_0, s_{\mathsf{M}_0})$, which implies that it is also final.

We define M as the image of the corecursor in M_0:

Definition 3.0.2.

$$\mathsf{M} :\equiv \sum_{m:\mathsf{M}_0} \left\| \sum_{((C,s_C):\mathsf{MCoalg}(A,B))} \sum_{(x:C)} \mathsf{corec}_{\mathsf{M}_0}(C, s_C, x) = m \right\|.$$

Our corecursor applies the corecursor on M_0 as the first component and stores its arguments in the second component:

Definition 3.0.3.

$$\mathsf{corec}_{\mathsf{M}} : \prod_{C:\mathcal{U}} \left(C \to (\sum_{a:A} B(a) \to C) \right) \to C \to \mathsf{M}$$
$$\mathsf{corec}_{\mathsf{M}}(C, s_C, x) :\equiv (\mathsf{corec}_{\mathsf{M}_0}(C, s_C, x), |((C, s_C), (x, \mathsf{refl}))|).$$

The interesting part is the definition of the destructor $s_{\mathsf{M}} : \mathsf{M} \to (\sum_{(a:A)} B(a) \to \mathsf{M})$. It has to satisfy the following computation rule.

$$s_{\mathsf{M}}(\mathsf{corec}_{\mathsf{M}}(C, s_C, x)) = (\mathsf{pr}_1(s_C(x)), \mathsf{corec}_{\mathsf{M}}(C, s_C) \circ \mathsf{pr}_2(s_C(x)))$$

Since C and s_C are stored in $\mathsf{corec}_{\mathsf{M}}(C, s_C, x)$, the definition of the destructor can return exactly the term on the right side of the computation rule. We just need to eliminate the propositional truncation before. For this, we need to eliminate into a mere proposition. We define this proposition as

$$P :\equiv \textstyle\sum((a, f) : \sum_{(a:A)} B(a) \to \mathsf{M}),\ s_{\mathsf{M}_0}(m_0) = (a, \mathsf{pr}_1 \circ f).$$

The first part of this type is exactly the result type of s_{M}. The second part is chosen to obtain a mere proposition. That it really is a mere propositon, can be seen with the equivalence

$$\begin{aligned} P &\equiv \textstyle\sum((a,f) : \sum_{(a:A)} B(a) \to \mathsf{M}),\ s_{\mathsf{M}_0}(m_0) = (a, \mathsf{pr}_1 \circ f) \\ &\simeq \textstyle\sum((a,p) : \sum_{(a)} \mathsf{label}_0(m_0) = a), \\ &\quad \textstyle\prod(b : B(a)),\ \sum((m_0', \_) : \sum_{(m_0')} \mathsf{arg}_0(m, p^{-1}{}_*(b)) = m_0'), \\ &\qquad \left\| \sum_{((C,s_C))} \sum_{(x)} \mathsf{corec}_{\mathsf{M}_0}(C, s_C, x) = m \right\|. \end{aligned}$$

Contractibility of the right side follows from repeated application of Facts 2.8.2, 2.8.4 and 2.8.5.

Definition 3.0.4. We define a helper-function $s'_{\mathsf{M}} : M \to P$ by truncation elimination as

$$s'_{\mathsf{M}}((m_0, |((C, s_C), (x, p))|)) :\equiv ((\mathsf{pr}_1(s_C(x)), \mathsf{corec}_{\mathsf{M}}(C, s_C) \circ \mathsf{pr}_2(s_C(x))), p).$$

where p is a the proof

$$\begin{aligned} s_{\mathsf{M}_0}(m_0) &= s_{\mathsf{M}_0}(\mathsf{corec}_{\mathsf{M}_0}(C, s_C, x)) \\ &= (\mathsf{pr}_1(s_C(x)), \mathsf{corec}_{\mathsf{M}_0}(C, s_C) \circ \mathsf{pr}_2(s_C(x))) \\ &\equiv (\mathsf{pr}_1(s_C(x)), \mathsf{pr}_1 \circ \mathsf{corec}_{\mathsf{M}}(C, s_C) \circ \mathsf{pr}_2(s_C(x))). \end{aligned}$$

Definition 3.0.5. We define the destructor $s_{\mathsf{M}} : \mathsf{M} \to \sum_{(a:A)} B(a) \to \mathsf{M}$ by

$$s_{\mathsf{M}}(m) :\equiv \mathsf{pr}_1(s'_{\mathsf{M}}(m)).$$

Remark 3.0.6. We have the computation rule for corecursion by definition:

$$s_M(\mathsf{corec}_{\mathsf{M}}(C, s_C, x)) \equiv (\mathsf{pr}_1(s_C(x)), \mathsf{corec}_{\mathsf{M}}(C, s_C) \circ \mathsf{pr}_2(s_C(x))).$$

Now we show that $(\mathsf{M}, s_{\mathsf{M}})$ is actually a final coalgebra by equality with $(\mathsf{M}_0, s_{\mathsf{M}_0})$. We start with an equivalence between both carriers.

Lemma 3.0.7.

$$\mathsf{M} \simeq \mathsf{M}_0$$

Proof. We define a function f from M to M_0 and show that it is an equivalence.

$$f((m_0, \_)) :\equiv m_0.$$

To prove that f is an equivalence, we define a quasi-inverse

$$g(m_0) :\equiv (m_0, |((\mathsf{M}_0, s_{\mathsf{M}_0}), (m_0, p))|)$$

Where $p : \mathsf{corec}_{\mathsf{M}_0}(\mathsf{M}_0, s_{\mathsf{M}_0}, m_0) = m_0$ is proven by uniqueness of corecursive functions on M_0. The function g is a right inverse to f by judgmental equality. It is also a left inverse because

$$\begin{aligned} g(f(m_0, y)) &\equiv g(m_0) \\ &\equiv (m_0, |((\mathsf{M}_0, s_{\mathsf{M}_0}), (m_0, \ldots))|) \\ &= (m_0, y) \end{aligned}$$

for all M_0 and y. The last line holds because the type of the second component is a mere proposition. Hence g is a quasi-inverse to f and thus f is an equivalence. □

Lemma 3.0.8.

$$(\mathsf{M}, s_{\mathsf{M}}) = (\mathsf{M}_0, s_{\mathsf{M}_0})$$

Proof. We use f and g as defined in Lemma 3.0.7. As equality on the first component we apply univalence to f. Then we have to show equivalence of both destructors over $\mathsf{ua}(f)$. This holds by

$$\begin{aligned} (\mathsf{ua}(f)_*(s_{\mathsf{M}}))(m_0) &= \mathsf{ua}(f)_*\Big(s_M(\mathsf{ua}(f)^{-1}{}_*(m_0))\Big) \\ &= \mathsf{ua}(f)_*(s_M(g(m_0))) \\ &\equiv \mathsf{ua}(f)_*(s_M((m_0, |((\mathsf{M}_0, s_{\mathsf{M}_0}), (m_0, \ldots))|))) \\ &\equiv \mathsf{ua}(f)_*((\mathsf{pr}_1(s_{\mathsf{M}_0}(m_0)), \mathsf{corec}_{\mathsf{M}}(\mathsf{M}_0, s_{\mathsf{M}_0}) \circ \mathsf{pr}_2(s_{\mathsf{M}_0}(m_0)))) \\ &= (\mathsf{pr}_1(s_{\mathsf{M}_0}(m_0)), \mathsf{ua}(f)_*(\mathsf{corec}_{\mathsf{M}}(\mathsf{M}_0, s_{\mathsf{M}_0}) \circ \mathsf{pr}_2(s_{\mathsf{M}_0}(m_0)))) \\ &= (\mathsf{pr}_1(s_{\mathsf{M}_0}(m_0)), f \circ \mathsf{corec}_{\mathsf{M}}(\mathsf{M}_0, s_{\mathsf{M}_0}) \circ \mathsf{pr}_2(s_{\mathsf{M}_0}(m_0))) \\ &\equiv (\mathsf{pr}_1(s_{\mathsf{M}_0}(m_0)), \mathsf{corec}_{\mathsf{M}_0}(\mathsf{M}_0, s_{\mathsf{M}_0}) \circ \mathsf{pr}_2(s_{\mathsf{M}_0}(m_0))) \\ &= (\mathsf{pr}_1(s_{\mathsf{M}_0}(m_0)), \mathsf{id} \circ \mathsf{pr}_2(s_{\mathsf{M}_0}(m_0))) \\ &\equiv s_{\mathsf{M}_0}(m_0). \end{aligned}$$

We apply Fact 2.4.5 in the first line and Fact 2.6.4 in the second sixth line. Line five is an application of Fact 2.4.6. In line eight we use uniqueness of corecursive functions. □

Lemma 3.0.9. *The coalgebra* $(\mathsf{M}, s_{\mathsf{M}})$ *is final.*

Proof. By Lemma 3.0.8 $(\mathsf{M}, s_{\mathsf{M}})$ is equal to $(\mathsf{M}_0, s_{\mathsf{M}_0})$. Thus finality of $(\mathsf{M}_0, s_{\mathsf{M}_0})$ implies finality of $(\mathsf{M}, s_{\mathsf{M}})$. □

This leads to the following general result.

Theorem 3.0.10. *For all types A and families $B : A \to \mathcal{U}$, there is a final coalgebra $(\mathsf{M}_{(a:A)}B(a), s_{\mathsf{M}_{(a:A)}B(a)})$ for the container functor $(\!|A \blacktriangleright B|\!)$. This coalgebra has a corecursion function*

$$\mathsf{corec}_{\mathsf{M}_{(a:A)}B(a)} : \prod_{C:\mathcal{U}} \left(C \to (\sum_{a:A} B(a) \to C)\right) \to C \to M$$

which satisfies the computation rule

$$s_{\mathsf{M}_{(a:A)}B(a)}(\mathsf{corec}_{\mathsf{M}_{(a:A)}B(a)}(C, s_C, x)) \equiv \Big(\mathsf{pr}_1(s_C(x)), \mathsf{corec}_{\mathsf{M}_{(a:A)}B(a)}(C, s_C) \circ \mathsf{pr}_2(s_C(x))\Big).$$

Proof. We use the type from Definition 3.0.2 as carrier and the function from Definition 3.0.5 as destructor. By Lemma 3.0.9 this coalgebra is final. As corecursor we use the function from Definition 3.0.3. The computation rule holds by Remark 3.0.6. □

In later chapters we will use the following functions to work with M-types.

Definition 3.0.11. Given a tree $m : \mathsf{M}a : AB(a)$, we define the label $\mathsf{label}(m) : A$ of m as.

$$\mathsf{label}(m) :\equiv \mathsf{pr}_1(s_{\mathsf{M}a:AB(a)}).$$

Definition 3.0.12. Given a tree $m : \mathsf{M}a : AB(a)$ and a subtree index $b : B(\mathsf{label}(m))$, we define the subtree $\mathsf{arg}(m, b) : \mathsf{M}a : AB(a)$ of m at index b as.

$$\mathsf{arg}(m, b) :\equiv \mathsf{pr}_2(s_{\mathsf{M}a:AB(a)})(b).$$

Fact 3.0.13. *For all $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, there is a pseudo-inverse*

$$\mathsf{sup} : \left(\sum_{a:A} B(a) \to \mathsf{M}_{a:A}B(a)\right) \to \mathsf{M}a : AB(a)$$

of the destructor $s_{\mathsf{M}a:AB(a)}$.

4 Construction of Strictly Positive Types

In this chapter we present a construction of nested inductive and coinductive types. First we define the syntax of strictly positive types and use it as a specification for types and functors. Then we show that containers are closed under all strictly positive type formers. It follows that we can build every strictly positive type as a container.

Definition 4.0.1. We define **strictly positive expressions** by the grammar

$$e_I, e'_I ::= K \mid x \mid e_I \times e'_I \mid e_I + e'_I \mid K \to e_I \mid \mu.e_{\mathsf{Option}(I)} \mid \nu.e_{\mathsf{Option}(I)}$$

where I and K are types and $x : I$. Hence the index of a metavariable indicates the type of free variables which the expression may contain. This means that we do not define a single type, but a family with the type of free variables as parameter. We will often ommit this index if it can be inferred from the context.

The last two rules represent inductive and coinductive types and work as binders. Therefore the inner expression may contain free variables from $\mathsf{Option}(I)$, which adds one variable none which we regard as the bound variable. We associate every other value $\mathsf{some}(x)$ with the variable x in the outer expression. This leads to a kind of De Bruijn notation in which every variable in a closed expression has the form $\mathsf{some}(\dots(\mathsf{some}(\mathsf{none})))$ and the number of nested constructor applications indicates the *distance* to the corresponding binder. With this in mind we define how strictly positive expressions serve as specification for types and functors. This specification depends on a context $\Gamma : I \to \mathcal{U}$ that binds all free variables or in the case of functors all free variables except one. First we need to define how we extend a context.

Definition 4.0.2. Given a context $\Gamma : I \to \mathcal{U}$ and a type X, the **extension** $\Gamma\,;;\,X : \mathsf{Option}(I) \to \mathcal{U}$ of Γ by X is defined by

$$\begin{aligned}(\Gamma\,;;\,X)(\mathsf{some}(i)) &:\equiv \Gamma(i)\\ (\Gamma\,;;\,X)(\mathsf{none}) &:\equiv X\end{aligned}$$

Definition 4.0.3. We define recursively what it means for a type to **realize** a strictly positive expression with free variables in I in a given context $\Gamma : I \to \mathcal{U}$:

- A type realizes the constant expression K in any context if it is equivalent to K.
- A type realizes the variable expression x in a context Γ if it is equivalent to $\Gamma(x)$.
- A type realizes the product expression $e \times e'$ in a context Γ if it is equivalent to a product $A_1 \times A_2$ and A_1 and A_2 realize e and e' respectively in context Γ.

- A type realizes the coproduct expression $e + e'$ in a context Γ if it is equivalent to a coproduct $A_1 + A_2$ and A_1 and A_2 realize e and e' respectively in context Γ.
- A type realizes the function type expression $K \to e$ in a context Γ if it is equivalent to a function type $K \to A$ and A realizes e in context Γ.
- A type realizes a fixed point expression $\mu.e$ in a context Γ if it is the initial algebra of a functor that realizes e in context Γ.
- A type realizes a fixed point expression $\nu.e$ in a context Γ if it is the final coalgebra of a functor that realizes e in context Γ.

A functor realizes a strictly positive expression e with free variables in $\mathsf{Option}(I)$ in a context $\Gamma : I \to \mathcal{U}$ if the type $F\,X$ realizes e in context $\Gamma\,;;\,X$ for all $X : \mathcal{U}$.

Most of the rules explicitly give the type that realizes an expression. If we had only those, the construction of strictly positive types would be easy. However, the specification for inductive and coinductive types is harder to satisfy. To build an initial algebra or a final coalgebra, we need more information about the functor. But if we know that it is a container functor, we can use the corresponding W- and M-type. For this reason we construct strictly positive types as containers.

Definition 4.0.4. An I-indexed container c **realizes** a strictly positive expression e if in every context Γ, the type $[\![c]\!]\,\Gamma$ realizes e.

Our goal is that given a strictly positive expression, we can recursively construct a container that realizes this expression. To every strictly positive type former, we devote one lemma in which we prove that container types are closed that particular type former.

Lemma 4.0.5. *Every* constant expression *is realized by a container.*

Proof. Given $K : \mathcal{U}$, we need a container $S \rhd P$ such that for all contexts Γ, we have $[\![S \rhd P]\!]\,\Gamma \simeq K$. If we set $P(i, s) :\equiv \mathbf{0}$, we generally have

$$\begin{aligned} [\![S \rhd P]\!]\,\Gamma &\equiv \sum_{(s:S)} \prod_{(i:I)} \mathbf{0} \to \Gamma(i) \\ &\simeq \sum_{(s:S)} \prod_{(i:I)} \mathbf{1} \\ &\simeq S \end{aligned} .$$

Hence it suffices to set $S :\equiv K$ and the container $S \rhd P$ realizes K. □

Lemma 4.0.6. *Every* variable expression *is realized by a container.*

Proof. Given $x : I$, we need a container $S \rhd P$ such that for all contexts Γ, we have

$[\![S \rhd P]\!]\,\Gamma \simeq \Gamma(x)$. If we set $S :\equiv \mathbf{1}$ and $P(i, \_) :\equiv x = i$, we have

$$\begin{aligned}
[\![S \rhd P]\!]\,\Gamma &\equiv \sum_{(\_:\mathbf{1})} \prod_{(i:I)} x = i \to \Gamma(i) \\
&\simeq \prod_{i:I} x = i \to \Gamma(i) \\
&\simeq \prod_{i:I} x = i \to \Gamma(x) \\
&\simeq (\sum_{i:I} x = i) \to \Gamma(x) \\
&\simeq \Gamma(x).
\end{aligned}$$

Hence the container $S \rhd P$ realizes x. ☐

Lemma 4.0.7. *Container types are closed under* product *i.e. if two expressions* e_1 *and* e_2 *are realized by containers then* $e_1 \times e_2$ *is realized by a container.*

Proof. Fix strictly positive expressions e_1 and e_2, and containers $S_1 \rhd P_1$ and $S_2 \rhd P_2$, that realize e_1 and e_2 respectively. We need a container $S \rhd P$ such that for all contexts Γ, we have $[\![S \rhd P]\!]\,\Gamma \simeq [\![S_1 \rhd P_1]\!]\,\Gamma \times [\![S_2 \rhd P_2]\!]\,\Gamma$. If we set $S :\equiv S_1 \times S_2$ and $P(i, (s_1, s_2,)) :\equiv P_1(i, s_1) + P_2(i, s_2)$, we have

$$\begin{aligned}
&[\![S \rhd P]\!]\,\Gamma \\
&\quad\equiv \sum_{((s_1,s_2):S_1 \times S_2)} \prod_{(i:I)} (P_1(i, s_1) + P_2(i, s_2)) \to \Gamma(i) \\
&\quad\simeq \sum_{((s_1,s_2):S_1 \times S_2)} \prod_{(i:I)} (P_1(i, s_1) \to \Gamma(i)) \times (P_2(i, s_2) \to \Gamma(i)) \\
&\quad\simeq \sum_{(s_1,s_2):S_1 \times S_2} \left(\prod_{i:I} P_1(i, s_1) \to \Gamma(i) \right) \times \left(\prod_{i:I} P_2(i, s_2) \to \Gamma(i) \right) \\
&\quad\simeq \left(\sum_{(s_1:S_1)} \prod_{(i:I)} P_1(i, s_1) \to \Gamma(i) \right) \times \left(\sum_{(s_2:S_2)} \prod_{(i:I)} P_2(i, s_2) \to \Gamma(i) \right) \\
&\quad\equiv [\![S_1 \rhd P_1]\!]\,\Gamma \times [\![S_2 \rhd P_2]\!]\,\Gamma.
\end{aligned}$$

Hence the container $S \rhd P$ realizes $e_1 \times e_2$. ☐

Lemma 4.0.8. *Container types are closed under* coproduct *i.e. if two expressions* e_1 *and* e_2 *are realized by containers then* $e_1 + e_2$ *is realized by a container.*

Proof. Fix strictly positive expressions e_1 and e_2, and containers $S_1 \rhd P_1$ and $S_2 \rhd P_2$, that realize e_1 and e_2 respectively. We need a container $S \rhd P$ such that for all contexts

Γ, we have $[\![S \rhd P]\!]\,\Gamma \simeq [\![S_1 \rhd P_1]\!]\,\Gamma + [\![S_2 \rhd P_2]\!]\,\Gamma$. If we set $S :\equiv S_1 + S_2$ and define P by

$$\begin{aligned} P(i, \mathsf{inl}(s)) &:\equiv P_1(i, s) \\ P(i, \mathsf{inr}(s)) &:\equiv P_2(i, s), \end{aligned}$$

we have

$$\begin{aligned} [\![S \rhd P]\!]\,\Gamma &\equiv \sum_{(s:S_1+S_2)} \prod_{(i:I)} P(i,s) \to \Gamma(i) \\ &\simeq \left(\sum_{(s:S_1)} \prod_{(i:I)} P(i, \mathsf{inl}(s)) \to \Gamma(i) \right) + \left(\sum_{(s:S_2)} \prod_{(i:I)} P(i, \mathsf{inr}(s)) \to \Gamma(i) \right) \\ &\equiv [\![S_1 \rhd P_1]\!]\,\Gamma + [\![S_2 \rhd P_2]\!]\,\Gamma. \end{aligned}$$

Hence the container $S \rhd P$ realizes $e_1 + e_2$. □

Lemma 4.0.9. *Container types are closed under* function types *with constant domain i.e. if an expressions e is realized by containers then $K \to e$ is realized by a container.*

Proof. Fix a strictly positive expression e and a container $S' \rhd P'$, that realizes e. We need a container $S \rhd P$ such that for all contexts Γ, we have $[\![S \rhd P]\!]\,\Gamma \simeq K \to [\![S' \rhd P']\!]\,\Gamma$. If we set $S :\equiv K \to S'$ and $P(i,s) :\equiv \sum_{(k:K)} P'(i, s(k))$, we have

$$\begin{aligned} [\![S \rhd P]\!]\,\Gamma &\equiv \sum_{(s:K\to S')} \prod_{(i)} \left(\sum_{k} P'(i, s(k)) \right) \to \Gamma(i) \\ &\simeq \sum_{(s:K\to S')} \prod_{(i)} \prod_{(k)} P'(i, s(k)) \to \Gamma(i) \\ &\simeq \sum_{(s:K\to S')} \prod_{(k)} \prod_{(i)} P'(i, s(k)) \to \Gamma(i) \\ &\simeq K \to \sum_{(s:S')} \prod_{(i)} P'(i, s) \to \Gamma(i) \\ &\equiv K \to [\![S' \rhd P']\!]\,\Gamma. \end{aligned}$$

Hence the container $S \rhd P$ realizes $K \to e$. □

Lemma 4.0.10. *Container types are closed under inductive fixed points i.e. if an expression e with free variables in* $\mathsf{Option}(I)$ *is realized by a container then $\mu.e$ is also realized by a container.*

Proof. Fix an expression e and an I-indexed container $S' \rhd P'$ that realizes e. We need a container $S \rhd P$ such that for every context Γ, we have a functor F that realizes e in Γ and $[\![S \rhd P]\!]\,\Gamma$ is an initial algebra of F. Fix a context Γ. We will define the functor F that realizes e in Γ. Then we will define $S \rhd P$ such that $[\![S \rhd P]\!]\,\Gamma$ is an initial algebra of F. The definition of $S \rhd P$ will not depend on Γ. Thus the condition is satisfied for all Γ and $S \rhd P$ realizes e.

We want the functor F to realize e in Γ. By definition this means that for all $X : \mathcal{U}$, the type $F\,X$ realizes e in $\Gamma\,;;\,X$. We already have the type $[\![S' \rhd P']\!](\Gamma\,;;\,X)$ that does this. Hence it suffices to define F such that $F\,X \simeq [\![S' \rhd P']\!](\Gamma\,;;\,X)$ for all X. We also want F to be a container functor because in this case we have an initial algebra. Therefore we transform $[\![S' \rhd P']\!](\Gamma\,;;\,X)$ into container form and use the result as definition for F.

$$\begin{aligned}
[\![S' \rhd P']\!](\Gamma\,;;\,X) &\equiv \sum_{(s:S')} \prod_{(i)} P'(i,s) \to (\Gamma\,;;\,X)(i) \\
&\simeq \sum_{s:S'} \left(\prod_{i} P'(\mathsf{some}(i), s) \to \Gamma(i) \right) \times P'(\mathsf{none}, s) \to X \\
&\simeq \textstyle\sum((s,\_) : \sum_{(s)} \prod_{(i)} P'(\mathsf{some}(i), s) \to \Gamma(i)),\ P'(\mathsf{none}, s) \to X \\
&\equiv (\!|\sum_{(s)} \prod_{(i)} P'(\mathsf{some}(i), s) \to \Gamma(i) \blacktriangleright P'(\mathsf{none}) \circ \mathsf{pr}_1 |\!)\, X
\end{aligned}$$

Let $F :\equiv (\!|\sum_{(s)} \prod_{(i)} P'(\mathsf{some}(i), s) \to \Gamma(i) \blacktriangleright P'(\mathsf{none}) \circ \mathsf{pr}_1 |\!)$. We know that the initial algebra of F is the W-type

$$\mathsf{W}((s,\_) : \textstyle\sum_{(s:S')} \prod_{(i)} P'(\mathsf{some}(i), s) \to \Gamma(i)),\ P'(\mathsf{none}, s).$$

We need to define the container $S \rhd P$ such that this type is the result of $[\![S \rhd P]\!]\,\Gamma$ and the definition does not depend on Γ. Again, we transform the W-type step by step into container form:

$$\begin{aligned}
&\mathsf{W}_{((s,\_):\sum_{(s:S')} \prod_{(i)} P'(\mathsf{some}(i),s) \to \Gamma(i))} P'(\mathsf{none}, s) \\
&\simeq \sum_{(w:\mathsf{W}_{(s:S)} P'(\mathsf{none},s))} \prod_{(s)} \mathsf{Addr}(s,w) \to \prod_{i} P'(\mathsf{some}(i), s) \to \Gamma(i) \\
&\simeq \sum_{(w:\mathsf{W}_{(s:S')} P'(\mathsf{none},s))} \prod_{(i)} \left(\sum_{s} \mathsf{Addr}(s,w) \times P'(\mathsf{some}(i), s) \right) \to \Gamma(i) \\
&\equiv [\![\mathsf{W}_{(s:S')} P'(\mathsf{none}, s) \rhd \lambda i.\, \lambda w.\, \sum_{s} \mathsf{Addr}(s,w) \times P'(\mathsf{some}(i), s)]\!]\,\Gamma.
\end{aligned}$$

In the first step we apply Lemma 4.0.11 that we prove next.

In summary, we have defined a functor F that realizes e in a given context and a container $S \rhd P$ that produces an initial algebra for F. The definition of the container does not depend on the context. Thus the $S \rhd P$ produces initial algebras for all contexts and realizes the expression $\mu.e$. □

Lemma 4.0.11. *For every $A_1 : \mathcal{U}$, $A_2 : A_1 \to \mathcal{U}$ and $B : A_1 \to \mathcal{U}$ there is an equivalence*

$$\mathsf{W}_{((a_1,\_):\sum_{(a_1:A_1)} A_2(a_1))} B(a_1) \simeq \sum_{(w:\mathsf{W}_{(a_1:A_1)} B(a_1))} \prod_{(a_1)} \mathsf{Addr}'(a_1, w) \to A_2(a_1)$$

We postpone the proof because it needs more preparation. Our intuition of the lemma is that on the left side we have a type of trees with pairs as labels, but the number of subtrees and with it the whole structure of the tree depends only on the first component of the labels. Thus we can split the tree into a tree that contains only the first component of each label and a function that assigns the second component to each node. For the remainder of the section fix a type A_1 and families $A_2, B : A_1 \to \mathcal{U}$. We abbreviate the left side of the equivalence as

$$\mathsf{W} :\equiv \mathsf{W}_{((a_1,\_):\sum_{(a_1:A_1)} A_2(a_1))} B(a_1).$$

For the right side we define

$$\mathsf{W}'_1 :\equiv \mathsf{W}_{(a_1:A_1)} B(a_1)$$
$$\mathsf{W}'_2(w) :\equiv \prod_{a_1} \mathsf{Addr}'(a_1, w) \to A_2(a_1).$$

Now we can write the equivalence as $\mathsf{W} \simeq \sum_{(w:\mathsf{W}'_1)} \mathsf{W}'_2(w)$. To prove Lemma 4.0.11, we will define functions $\mathsf{dec} : (\sum_{(w:\mathsf{W}'_1)} \mathsf{W}'_2(w)) \to \mathsf{W}$ and $\mathsf{undec} : \mathsf{W} \to \sum_{(w:\mathsf{W}'_1)} \mathsf{W}'_2(w)$ and show that undec is the pseudo-inverse of dec.

Definition 4.0.12.

$$\begin{aligned} \mathsf{dec} &: \left(\textstyle\sum_{(w:\mathsf{W}'_1)} \mathsf{W}'_2(w)\right) \to \mathsf{W} \\ \mathsf{dec}((\mathsf{sup}(a_1, f), g)) &:\equiv \mathsf{sup}((a_1, g(a_2, \mathsf{root\_addr})), \\ &\qquad \lambda b.\, \mathsf{dec}((f(b), g \circ \mathsf{subtree\_addr}(b)))) \end{aligned}$$

Definition 4.0.13. We define undec component-wise with components

$$\begin{aligned} \mathsf{undec}_1 &: \mathsf{W} \to \mathsf{W}'_1 \\ \mathsf{undec}_1(\mathsf{sup}((a_1, \_), f)) &:\equiv \mathsf{sup}(a_1, \mathsf{undec}_1 \circ f) \\ \mathsf{undec}_2 &: \textstyle\prod(w : \mathsf{W}),\ \mathsf{W}'_2(\mathsf{undec}_1(w)) \\ \mathsf{undec}_2(\mathsf{sup}((\_, a_2), \_), \mathsf{root\_addr}) &:\equiv a_2 \\ \mathsf{undec}_2(\mathsf{sup}(\_, f), \mathsf{subtree\_addr}(b, \mathit{addr}')) &:\equiv \mathsf{undec}(f(b), \mathit{addr}'). \end{aligned}$$

We combine them to

$$\begin{aligned} \mathsf{undec} &: \mathsf{W} \to \textstyle\sum_{(w:\mathsf{W}'_1)} \mathsf{W}'_2(w) \\ \mathsf{undec}(w) &:\equiv (\mathsf{undec}_1(w), \mathsf{undec}_2(w)). \end{aligned}$$

For the proof, that dec and undec are mutually inverse, we will use an encoding of paths in W-types, that simplifies our work significantly.

Definition 4.0.14. Given a type A, a family $B : A \to \mathcal{U}$ and inhabitants $\mathsf{sup}(a_1, f_1)$ and $\mathsf{sup}(a_2, f_2)$ of $\mathsf{W}_{(a:A)} B(a)$, we define

$$\mathsf{sup}(a_1, f_1) \approx_W \mathsf{sup}(a_2, f_2) :\equiv \sum_{(p:a_1 = a_2)} \prod_{(b)} f_1(b) \approx_W f_w(p_*(b))$$

Lemma 4.0.15. *For $A : \mathcal{U}$, $B : A \to \mathcal{U}$ and $w_1, w_2 : \mathsf{W}_{(a:A)} B(a)$ there is an equivalence*

$$(w_1 = w_2) \simeq (w_1 \approx_W w_2).$$

We call it encode *and its pseudo-inverse* decode*.*

Proof. Fix some $w_1, w_2 : \mathsf{W}_{(a:A)} B(a)$. We will use the fact that $w_1 = w_2$ has the same recursive structure as $w_1 \approx_W w_2$ to do a proof by induction on w_1. For the inductive step assume that $w_1 \equiv \mathsf{sup}(a_1, f_1)$ and $w_2 \equiv \mathsf{sup}(a_2, f_2)$. Then we have

$$\begin{aligned}
\mathsf{sup}(a_1, f_1) = \mathsf{sup}(a_2, f_2) &\simeq \sum_{p:a_1=a_2} p_*(f_1) = f_2 \\
&\simeq \sum_{p:a_1=a_2} f_1 = {p^{-1}}_*(f_2) \\
&\simeq \sum_{(p:a_1=a_2)} \prod_{(b)} f_1(b) = ({p^{-1}}_*(f_2))(b) \\
&\simeq \sum_{(p:a_1=a_2)} \prod_{(b)} f_1(b) = f_2(p_*(b)) \\
&\simeq \sum_{(p:a_1=a_2)} \prod_{(b)} f_1(b) \approx_W f_2(p_*(b)) \\
&\equiv \mathsf{sup}(a_1, f_1) \approx_W \mathsf{sup}(a_2, f_2).
\end{aligned}$$

In the second to last step we apply the induction hypothesis. □

Now we prove that undec is right inverse and left inverse to dec. The first proof is simple.

Lemma 4.0.16. *For all $w : \mathsf{W}$ we have*

$$w = \mathsf{dec}(\mathsf{undec}(w)).$$

Proof. By induction on w. □

The second part is harder to prove. We will show it seperately for both components.

Lemma 4.0.17. *For all $w : \mathsf{W}'_1$ and $g : \mathsf{W}'_2(w)$ we have*

$$w \approx_W \mathsf{undec}_1(\mathsf{dec}(w, g)).$$

Proof. We do a prove by induction on w for arbitrary g. For the inductive step assume that $w \equiv \mathsf{sup}(a_1, f)$. We have

$$\begin{aligned}
&\mathsf{sup}(a_1, f) \\
&\qquad \approx_W \mathsf{sup}(a_1, \lambda b.\, \mathsf{undec}_1(\mathsf{dec}((f(b), g \circ \mathsf{subtree\_addr}(b))))) \\
&\qquad \equiv \mathsf{undec}_1(\mathsf{sup}((a_1, g(\mathsf{root\_addr})), \lambda b.\, \mathsf{dec}((f(b), g \circ \mathsf{subtree\_addr}(b))))) \\
&\qquad \equiv \mathsf{undec}_1(\mathsf{dec}(\mathsf{sup}(a_1, f), g)).
\end{aligned}$$

The resulting proof term for the inductive step consists of the identity path in the first component and an application of the induction hypothesis in the second component. The proof term for the whole lemma is given by the recursive definition

$$\mathsf{undec_1\_dec}(\mathsf{sup}(a_1, f), g) :\equiv (\mathsf{refl}_{a_1}, \lambda b.\, \mathsf{undec_1\_dec}(f(b), g \circ \mathsf{subtree\_addr}(b)).$$

□

Corollary 4.0.18. *For all $w : \mathsf{W}'_1$ and $g : \mathsf{W}'_2(w)$ we have*

$$w = \mathsf{undec_1}(\mathsf{dec}(w, g)).$$

For the proof of the equality in the second component, we will work with a transport over the equality defined by Corollary 4.0.18. It will be usefull, to have a recursive description of transport of addresses over paths that were created by decode.

Definition 4.0.19. For a type A and a family $B : A \to \mathcal{U}$, we define

$$\mathsf{transport\_addr} : \prod_{w_1, w_2 : \mathsf{W}_{(a:A)} B(a)} (w_1 \approx_W w_2) \to \prod_{a:A} \mathsf{Addr}'(a, w_1) \to \mathsf{Addr}'(a, w_2)$$

$$\begin{aligned}
&\mathsf{transport\_addr}(\mathsf{sup}(a_1, f_1), \mathsf{sup}(a_2, f_2), (p, q), a_1, \mathsf{root\_addr}) \\
&\qquad :\equiv \mathsf{transport}^{\mathsf{Addr}(-, \mathsf{sup}(a_2, f_2))}(p^{-1}, \mathsf{root\_addr}) \\
&\mathsf{transport\_addr}(\mathsf{sup}(a_1, f_1), \mathsf{sup}(a_2, f_2), (p, q), a, \mathsf{subtree\_addr}(b, addr)) \\
&\qquad :\equiv \mathsf{subtree\_addr}(p_*(b), \mathsf{transport\_addr}(f_1(b), f_2(p_*(b)), q(b), a, addr)
\end{aligned}$$

What $\mathsf{transport\_addr}$ does is that it recursively transports the subtree indices but doesnt change the structure of an address in any other way. The following lemma shows that this is indeed how transport of addresses behaves.

Lemma 4.0.20. *For all types A, families $B : A \to \mathcal{U}$, trees $w_1, w_2 : \mathsf{W}_{(a:A)} B(a)$, encoded paths $p : w_1 \approx_W w_2$, labels $a : A$ and addresses $addr : \mathsf{Addr}(a, w_1)$ we have*

$$\mathsf{transport}^{\mathsf{Addr}(a)}(\mathsf{decode}(p), addr) = \mathsf{transport\_addr}(w_1, w_2, p, a, addr).$$

Proof. We show the equivalent statement that

$$\mathsf{transport}^{\mathsf{Addr}(a)}(p, addr) = \mathsf{transport\_addr}(w_1, w_2, \mathsf{encode}(p), a, addr)$$

for all paths $q : w_1 = w_2$ by path induction on q and induction on $addr$. □

Lemma 4.0.21. *Given $w : \mathsf{W}'_1$ and $g : \mathsf{W}'_2(w)$, let $p : w = \mathsf{undec_1}(\mathsf{dec}((w, g)))$ be the path from Corollary 4.0.18. Then we have*

$$p_*(g) = \mathsf{undec_2}(\mathsf{dec}(w, g)).$$

Proof. We move the transport to the right side and apply functional extensionality to show the equivalent statement

$$g(addr) = {p^{-1}}_*(\mathsf{undec}_2(\mathsf{dec}(w,g)))(addr)$$

for arbitrary but fixed $a : A_1$ and $addr : \mathsf{Addr}(a,w)$. We do this by the following calculation.

$$\begin{aligned}
&{p^{-1}}_*(\mathsf{undec}_2(\mathsf{dec}(w,g)))(addr)\\
&\quad = \mathsf{undec}_2(\mathsf{dec}(w,g), p_*(addr))\\
&\quad \equiv \mathsf{undec}_2(\mathsf{dec}(w,g), \mathsf{decode}(\mathsf{undec}_1\_\mathsf{dec}(w,g))_*(addr))\\
&\quad = \mathsf{undec}_2(\mathsf{dec}(w,g), \mathsf{transport\_addr}(\mathsf{decode}(\mathsf{undec}_1\_\mathsf{dec}(w,g)), addr)\\
&\quad = g(addr)
\end{aligned}$$

In the second to last step we apply Lemma 4.0.20. We prove the last step by induction on *addr* for arbitrary g.

The inductive case for the root address holds by reflexivity.

$$\begin{aligned}
&\mathsf{undec}_2(\mathsf{dec}(w,g), \mathsf{transport\_addr}(\mathsf{decode}(p), \mathsf{root\_addr}))\\
&\quad \equiv \mathsf{undec}_2(\mathsf{dec}(w,g), \mathsf{root\_addr})\\
&\quad \equiv g(\mathsf{root\_addr}).
\end{aligned}$$

The inductive case for subtree addresses consists of a single application of the inductive hypothesis.

$$\begin{aligned}
&\mathsf{undec}_2(\mathsf{dec}(w,g), \mathsf{transport\_addr}(\mathsf{decode}(p), \mathsf{subtree\_addr}(b, addr))\\
&\quad \equiv \mathsf{undec}_2(\mathsf{dec}(w,g),\\
&\qquad\qquad \mathsf{subtree\_addr}(b, \mathsf{transport\_addr}(\mathsf{decode}(\mathsf{pr}_2(p)(b)), addr)))\\
&\quad \equiv \mathsf{undec}_2(\mathsf{arg}(\mathsf{dec}((w,g)), b),\\
&\qquad\qquad \mathsf{transport\_addr}(\mathsf{decode}(\mathsf{pr}_2(p)(b)), addr))\\
&\quad \equiv \mathsf{undec}_2(\mathsf{dec}(f(b), g \circ \mathsf{subtree\_addr}(b)),\\
&\qquad\qquad \mathsf{transport\_addr}(\mathsf{decode}(\mathsf{pr}_2(p)(b)), addr))\\
&\quad = g(\mathsf{subtree\_addr}(b, addr)).
\end{aligned}$$

□

Corollary 4.0.22. *The function* undec *is left inverse to* dec*.*

Proof. By Corollary 4.0.18 and Lemma 4.0.21. □

With this corollary we have everything that we need to prove the equivalence.

Proof of Lemma 4.0.11. We have to show $\mathsf{W} \simeq \sum_{(w:\mathsf{W}_1')} \mathsf{W}_2'(w)$. There is a function $\mathsf{dec} : (\sum_{(w:\mathsf{W}_1')} \mathsf{W}_2'(w)) \to \mathsf{W}$ and a function $\mathsf{undec} : \mathsf{W} \to \sum_{(w:\mathsf{W}_1')} \mathsf{W}_2'(w)$ which is a pseude-inverse by Corollary 4.0.22 and Lemma 4.0.16. □

Now that we have finished the construction of inductive-fixed-point containers, the only case that is left are coinductive fixed points. The construction is analogous to that of inductive fixed points. Instead of W-types we use M-types, instead of recursion we use corecursion and instead of induction for equality proofs we apply the use the uniqueness of corecursive functions. The only big difference is that we don't have a recursive encoding of equality between elements of an M-type, which makes some manual rewriting necessary. We don't present the full construction here and only give the result.

Fact 4.0.23. *Container types are closed under coinductive fixed points i.e. if an expression e with free variables in $\mathsf{Option}(I)$ is realized by a container then $\nu.e$ is also realized by a container.*

We conclude the chapter with a theorem that summarizes our work.

Theorem 4.0.24. *Every strictly positive expression is realized by a container. In particular if a strictly positive expression is closed i.e. the type of free variables is $\mathsf{0}$ then it is realized by a type.*

Proof. We construct the container by recursion on the expression. For the different cases we apply Lemmas 4.0.5 to 4.0.10 and Fact 4.0.23. If the expression is closed, we obtain the realizing type by application of the container on the empty context. $\square$

5 W-Types

In this chapter we present a construction of W-types as subtypes of the corresponding M-types. For the remainder of the chapter fix a type A and a family $B : A \to \mathcal{U}$ and abbreviate $\mathsf{M}_{(a:A)}B(a)$ to M. First we will construct an initial algebra $(\mathsf{W}', s'_{\mathsf{W}})$ for the functor $(\!|A \blacktriangleright B|\!)$. Then we will refine it, to obtain one that satisfies the computation rule for the non-dependent recursor judgmentally.

5.1 Construction of W-Types

The basic idea of the construction is based on the intuition that W-types and M-types both describe trees, with the difference that elements of a W-type need to be well-founded. If we have a definition of well-foundedness, it is straight forward to define W' as a subtype of M. Well-foundedness would be easy to define as an inductive predicate. Without a mechanism for general inductive definitions we need an impredicative encoding of well-foundedness.

Definition 5.1.1. A tree $m : \mathsf{M}$ is **well-founded** if the induction principle holds on m for all mere predicates:

$$\mathsf{isWf}(m) :\equiv \prod_{P:\mathsf{M}\to\mathsf{Prop}} \left(\prod_{(a:A)} \prod_{(f:B(a)\to\mathsf{M})} \left(\prod_{b:B(a)} P(f(b)) \right) \to P(\mathsf{sup}(a,f)) \right) \to P(m).$$

It is important that we allow only mere predicates for P, because now the codomain of isWf is a mere proposition, which makes isWf itself into a mere predicate. Hence the following definition yields a subtype.

Definition 5.1.2. We define W' as the subtype of well-founded elements in M:

$$\mathsf{W}' :\equiv \sum_{m:\mathsf{M}} \mathsf{isWf}(m).$$

Now we define a constructor and two destructors on W'.

Definition 5.1.3. We define the **constructor** $\mathsf{sup}' : \prod_{(a:A)}(B(a) \to \mathsf{W}') \to \mathsf{W}'$ in two components. Let $a : A$ and $f : B(a) \to \mathsf{W}'$ be given. For the first component we apply the constructor of M. For the second component we now need to prove well-foundedness of the resulting tree $\mathsf{sup}(a, \mathsf{pr}_1 \circ f)$. Fix a mere predicate $P : \mathsf{M} \to \mathsf{Prop}$ and a step function $\mathit{step} : \prod_{(a:A)} \prod_{(f:B(a)\to\mathsf{M})} \left(\prod_{(b:B(a))} P(f(b))\right) \to P(\mathsf{sup}(a,f))$. To show $P(\mathsf{sup}(a, \mathsf{pr}_1 \circ f))$, we apply *step* and we are left with the proof obligation $P(\mathsf{pr}_1(f(b)))$ for an arbitrary $b : B(a)$. This is proven by induction on $\mathsf{pr}_1(f(b))$, which means, we apply the proof of well-foundedness $\mathsf{pr}_2(f(b))$.

Definition 5.1.4. We define a **label destructor** $\mathsf{label} : \mathsf{W}' \to A$ by an application of the label destructor of the M-type.

Definition 5.1.5. For the definition of a subtree destructor

$$\mathsf{arg}(w, b) : \prod_{w:\mathsf{W}'} B(\mathsf{label}(w)) \to \mathsf{W}'$$

we apply the argument destructor of the M-type to obtain an element of M that we return as first compnent. Now we have to show that the resulting subtree $\mathsf{arg}(\mathsf{pr}_1(w), b)$ is again well-founded. We fix some predicate P and step function *step* and have to prove $P(\mathsf{arg}(\mathsf{pr}_1(w), b))$. We do this by generalization to $\prod_{(b:B(\mathsf{label}(m)))} P(\mathsf{arg}(\mathsf{pr}_1(w), b))$ and induction on m.

Fact 5.1.6. *The constructor and destructors on* W' *are mutually pseudo-inverse:*

- *For all* $w : \mathsf{W}'$ *we have*
$$\mathsf{sup}'(\mathsf{label}(w), \mathsf{arg}(w)) = w.$$
- *For all* $a : A$ *and* $f : B(a) \to \mathsf{W}'$ *we have*
$$\Big(\mathsf{label}(\mathsf{sup}'(a, f)), \mathsf{arg}(\mathsf{sup}'(a, f))\Big) = (a, f).$$

Now we want to show initiality of W', which is defined as the contractibility of the type of homomorphisms from W' to all other $(\!|A \blacktriangleright B|\!)$-algebras. For the remainder of the section we fix such an algebra (C, s_C) and show contractibility of homomorphisms from W' to (C, s_C). As an intermediate step we show contractibility of what we call local morphisms. The type of morphisms local to some $w : \mathsf{W}'$ is intended to contain morphisms that are defined only on the subtrees of w. To simplify proofs we do not really use the type of subtrees in the formal definition, but the type of addresses in w. Every address will represent the subtree at that position. The difference is that a single subtree can be found at many addresses, hence the type of addresses might be larger. We may still talk about subtrees in informal explanations, as it better reflects the intuition behind our approach.

Definition 5.1.7. We define the type of **local morphisms** from W' to (C, s_C) at position w similar to that of regular homorphisms from W' to (C, s_C):

$$\mathsf{LHom}(w) :\equiv \sum_{(h:\mathsf{Addr}(w)\to C)} \prod_{(addr)} s_C(\mathsf{label\_at}(w, addr), h \circ \mathsf{extend\_addr}(addr)) = h(addr).$$

Note that the computation rule for regular homomorphisms tells us how the morphism behaves on the result of constructor applications while we talk about destructor applications in the definition of local morphisms. This is because we have no good way to express constructor application on addresses. But since the constructor and destructors are mututally inverse, we could express the computation rule for regular homomorphisms in the same form that we used here.

Lemma 5.1.8. *For all* $w : \mathsf{W}'$ *we have*

$$\mathsf{LHom}(w) \simeq \prod_{b:B(\mathsf{label}(w))} \mathsf{LHom}(\mathsf{arg}(w,b)).$$

Proof. The lemma holds by

$$\begin{aligned}
&\mathsf{LHom}(\mathsf{sup}(a,f))\\
&\quad\equiv \sum_{(h)} \prod_{(addr)} s_C(\mathsf{label\_at}(w, addr), h \circ \mathsf{extend\_addr}(addr)) = h(addr)\\
&\quad\simeq \sum_{(h)} \prod_{(addr)}\\
&\qquad (s_C(\mathsf{label\_at}(w, \mathsf{root\_addr}), h \circ \mathsf{extend\_addr}(\mathsf{root\_addr})) = h(\mathsf{root\_addr}))\\
&\qquad \times (\prod_{(b:B(\mathsf{label}(w)))} \prod_{(addr:\mathsf{Addr}(\mathsf{arg}(w,b)))}\\
&\qquad s_C(\mathsf{label\_at}(w, \mathsf{subtree\_addr}(b, addr), h \circ \mathsf{extend\_addr}(\mathsf{subtree\_addr}(b, addr)) =\\
&\qquad\qquad h(\mathsf{subtree\_addr}(b, addr))\\
&\quad\simeq \sum_{(c:C)} \sum_{(h:\prod_{(b:B(\mathsf{label}(w)))} \mathsf{Addr}(\mathsf{arg}(w,b)) \to C)}\\
&\qquad (s_C(\mathsf{label}(w), \lambda b.\, h(b, \mathsf{root\_addr})) = c)\\
&\qquad \times (\prod_{(b)} \prod_{(addr)}\\
&\qquad s_C(\mathsf{label\_at}(\mathsf{arg}(w,b), addr), \lambda b'.\, h(b, \mathsf{extend\_addr}(addr, b'))) =\\
&\qquad\qquad h(b, addr))\\
&\quad\simeq \sum_{(h:\prod_{(b:B(\mathsf{label}(w)))} \mathsf{Addr}(\mathsf{arg}(w,b)) \to C)} \prod_{(b)} \prod_{(addr)}\\
&\qquad s_C(\mathsf{label\_at}(\mathsf{arg}(w,b), addr), \lambda b'.\, h(b, \mathsf{extend\_addr}(addr, b'))) = h(b, addr)\\
&\quad\simeq \prod_{(b:\mathsf{label}(w))} \sum_{(h:\mathsf{Addr}(\mathsf{arg}(w,b)) \to C)} \prod_{(addr)}\\
&\qquad s_C(\mathsf{label\_at}(\mathsf{arg}(w,b), addr), \lambda b'.\, h(b, \mathsf{extend\_addr}(addr, b'))) = h(b, addr)\\
&\quad\simeq \prod_{b:B(a)} \mathsf{LHom}(f(b)).
\end{aligned}$$

□

Lemma 5.1.9. *For all $w : \mathsf{W}'$ the type $\mathsf{LHom}(w)$ of local morphisms at w is contractible.*

Proof. Assume w is a pair $(m, wf) : \mathsf{W}$. We apply wf to show that for all $wf' : \mathsf{isWf}(m)$ the type $\mathsf{LHom}((m, wf'))$ is contractible by well-founded induction on m. For this application, we need to prove that the goal is a mere proposition, which holds by Fact 2.8.6. But we also need to use propositional resizing on the goal because it contains the well-foundedness predicate and $wf : \mathsf{isWf}(m)$, which means that wf can only be instanciated with predicates from a smaller universe. For the inductive step fix $wf' : \mathsf{isWf}(m)$. We use the equivalence

$$\begin{aligned}\mathsf{LHom}((m, wf')) &\simeq \prod_{b : B(\mathsf{label}((m,wf')))} \mathsf{LHom}(\mathsf{arg}((m, wf'), b)) \\ &\equiv \prod_{b : B(\mathsf{label}((m,wf')))} \mathsf{LHom}((\mathsf{arg}(m, b), \ldots))\end{aligned}$$

from Lemma 5.1.8 which makes it sufficient to show $\mathsf{isContr}(\mathsf{LHom}((\mathsf{arg}(m, b), \ldots)))$ for all $b : \mathsf{label}(m)$. This holds by the inductive hypothesis. □

Corollary 5.1.10. *The type $\prod_{(w:\mathsf{W}')} \mathsf{LHom}(w)$ is contractible.*

Now we have contractibility of local homomorphisms and need to show that this implies contractibility of regular homomorphisms, i.e. inhabitants of $\mathsf{AlgHom}((A, s_A), (C, s_C))$. We will work with an alternative definition of W-homomorphisms, which is more similar to that of local morphisms and thus simplifies parts of the proof.

Definition 5.1.11.

$$\mathsf{WHom}' :\equiv \sum_{(h:\mathsf{W}' \to X)} \prod_{(w)} s_C(\mathsf{label}(w), h \circ \mathsf{arg}(w)) = h(w).$$

This definition is justified by the following lemma.

Lemma 5.1.12.

$$\mathsf{WHom}' \simeq \mathsf{WHom}((\mathsf{W}', \mathsf{sup}'), (C, s_C)).$$

Proof. The lemma holds by

$$\begin{aligned}\mathsf{WHom}' &\equiv \sum_{(h:\mathsf{W}' \to X)} \prod_{(w)} s_C(\mathsf{label}(w), h \circ \mathsf{arg}(w)) = h(w) \\ &\simeq \sum_{(\mathsf{W}' \to X)} \prod_{(a)} \prod_{(f)} h(\mathsf{sup}'(a, f)) = s_C(a, h \circ f) \\ &\equiv \mathsf{WHom}((\mathsf{W}', \mathsf{sup}'), (C, s_C)).\end{aligned}$$

In the second line we use that the constructor and destructors are mutually inverse by Fact 5.1.6 and hence form an equivalence. □

We will show that $\mathsf{WHom}' \simeq \prod_{(w)} \mathsf{LHom}(w)$. Then Corollary 5.1.10 implies contractibility of WHom'. We prove the equivalence with two mutually inverse functions $\Phi : \mathsf{WHom}' \to \prod_{(w)} \mathsf{LHom}(w)$ and $\Psi : \left(\prod_{(w)} \mathsf{LHom}(w)\right) \to \mathsf{WHom}'$.

Definition 5.1.13. We define the first component of Φ by

$$\Phi_1((h,\beta), w, addr) :\equiv h(\mathsf{subtree\_at}(w, addr)).$$

For the second component we prove the computation rule:

$$\begin{aligned}
&\Phi_1((h,\beta), w, addr)\\
&\quad\equiv h(\mathsf{subtree\_at}(w, addr))\\
&\quad= s_C(\mathsf{label\_at}(w, addr), h \circ \mathsf{arg}(\mathsf{subtree\_at}(w, addr)))\\
&\quad= s_C(\mathsf{label\_at}(w, addr), h \circ \mathsf{subtree\_at}(w) \circ \mathsf{extend\_addr}(addr))\\
&\quad\equiv s_C(\mathsf{label\_at}(w, addr), h' \circ \mathsf{extend\_addr}(addr)).
\end{aligned}$$

On the second line we apply the computation rule β and on the third line we prove $\mathsf{arg}(\mathsf{subtree\_at}(w, addr)) = \mathsf{subtree\_at}(w) \circ \mathsf{extend\_addr}(addr)$ by induction on $addr$.
The resulting proof term is

$$\Phi_2((h,\beta), w, addr) :\equiv \beta(\mathsf{subtree\_at}(w, addr)) \cdot \mathsf{subtree\_at\_extend\_addr}(w, addr).$$

We combine both components to the definition

$$\Phi((h,\beta), w) :\equiv \Big(\Phi_1((h,\beta), w), \Phi_2((h,\beta), w)\Big).$$

All that we need to know about the second component of Φ, is expressed in the following lemma.

Lemma 5.1.14. *For all* $(h,\beta) : \mathsf{WHom}'$ *and* $w : \mathsf{W}'$ *we have*

$$\mathsf{pr}_2(\Phi((h,\beta), w))(\mathsf{root\_addr}) = \beta(w).$$

Proof. By the judgmental equality $\mathsf{subtree\_at\_extend\_addr}(w, \mathsf{root\_addr}) \equiv \mathsf{refl}$. □

For the definition of Ψ we will have a family of local morphisms and our task is to combine them to a single regular morphism. For this we will need the fact that all local morphisms are compatible in the sense that they agree on common subtrees. To formulate this we will need the restriction of local morphisms to subtrees.

Definition 5.1.15. For a tree $w : \mathsf{W}'$, a local morphism $H : \mathsf{LHom}(w)$ and a subtree index $b : B(\mathsf{label}(w))$, we define the **restriction of** H to the subtree at index b as

$$\begin{aligned}
H\big|_b &: \mathsf{LHom}(\mathsf{arg}(w, b))\\
H\big|_b &:\equiv (\mathsf{pr}_1(H) \circ \mathsf{subtree\_addr}(b),\\
&\qquad \mathsf{pr}_2(H) \circ \mathsf{subtree\_addr}(b)).
\end{aligned}$$

Definition 5.1.16. We define the first component of Ψ as

$$\Psi_1(H, w) :\equiv \mathsf{pr}_1(H(w))(\mathsf{root\_addr}).$$

For the definition of the second component we would like to apply the computation rule from one of the local morphisms. The problem is that we defined Ψ_1 from different local morphisms. Hence we need to fill a gap here and show that those morphisms agree on the relevant subtrees. That is where the restriction comes in. The computation rule for Ψ_1 holds by

$$\begin{aligned}\Psi_1(H,w) &\equiv \mathsf{pr}_1(H(w))(\mathsf{root\_addr})\\ &= s_C(\mathsf{label}(w), \mathsf{pr}_1(H(w)) \circ \mathsf{extend\_addr}(\mathsf{root\_addr}))\\ &\equiv s_C(\mathsf{label}(w), \lambda b.\, \mathsf{pr}_1(H(w)) \circ \mathsf{subtree\_addr}(b, \mathsf{root\_addr}))\\ &\equiv s_C(\mathsf{label}(w), \lambda b.\, \mathsf{pr}_1(H(w)\big|_b)(\mathsf{root\_addr}))\\ &= s_C(\mathsf{label}(w), \lambda b.\, \mathsf{pr}_1(H(\mathsf{arg}(w,b)))(\mathsf{root\_addr}))\\ &\equiv s_C(\mathsf{label}(w), \Psi_1(H) \circ \mathsf{arg}(w)).\end{aligned}$$

In the second line we apply the computation rule $\mathsf{pr}_2(H(w))$. In the fourth line we use that $H(w)\big|_b = H(\mathsf{arg}(w,b))$, because the type of local recursors is contractible. The resulting proof term is

$$\Psi_2(H,w) :\equiv \mathsf{pr}_2(H(w))(\mathsf{root\_addr}) \cdot \kappa(\lambda b.\, \mathsf{path\_contr}(H\big|_b, H(\mathsf{arg}(w,b)))).$$

where the function

$$\kappa(p) :\equiv \mathsf{ap}_{s_C(\mathsf{label}(w),-)}(\mathsf{funext}(\lambda b.\, \mathsf{ap}_{\mathsf{pr}_1(-)(\mathsf{root\_addr})}(p(b))))$$

defines the context in which we apply the second rewrite step.

We combine those two components to the definition

$$\Psi(H) :\equiv (\Psi_1(H), \Psi_2(H)).$$

What is left to show, is that Φ and Ψ are mutually inverse. The first direction is trivial.

Lemma 5.1.17. *The function Ψ is right inverse to Φ i.e. for all families of local morphisms $H : \prod_{(w)} \mathsf{LHom}(w)$ we have*

$$\Phi(\Psi(H)) = H.$$

Proof. By contractibility of local morphisms. □

For the other direction we need to replace the resulting path from an application of path_contr by a path with known behavior. This is possible with the following fact.

Fact 5.1.18. *Let x and y be inhabitants of a contractible type. Then the type of paths $x = y$ is contractible.*

Lemma 5.1.19. *The function Ψ is left inverse to Φ i.e. for all homomorphisms (h,β) :* WHom *we have*

$$\Psi\Big(\Phi((h,\beta))\Big) = (h,\beta).$$

Proof. The equality holds by

$$\begin{aligned}
&\Psi(\Phi((h,\beta)))\\
&\quad\equiv \Psi(\lambda w.\,(h \circ \mathsf{subtree\_addr}(w), \Phi_2((h,\beta),w)))\\
&\quad\equiv (\lambda w.\, h \circ \mathsf{subtree\_at}(w, \mathsf{root\_addr}), \Psi_2(\Phi((h,\beta))))\\
&\quad\equiv (h, \Psi_2(\Phi((h,\beta))))\\
&\quad= (h,\beta).
\end{aligned}$$

The last step consists of functional extensionality applied to the following computation:

$$\begin{aligned}
&\Psi_2(\Phi((h,\beta)),w)\\
&\quad\equiv \Phi_2((h,\beta),w)(\mathsf{root\_addr})\\
&\qquad\cdot \kappa(\lambda b.\,\mathsf{path\_contr}(\Phi((h,\beta))'(w,b), \Phi((h,\beta), \mathsf{arg}(w,b))))\\
&\quad= \Phi_2((h,\beta),w)(\mathsf{root\_addr}) \cdot \kappa(\lambda b.\,\mathsf{refl})\\
&\quad\equiv \beta(w) \cdot \kappa(\lambda b.\,\mathsf{refl})\\
&\quad= \beta(w)
\end{aligned}$$

For the second equality we use Fact 5.1.18 to replace the path

$$\mathsf{path\_contr}(\Phi((h,\beta))'(w,b), \Phi((h,\beta), \mathsf{arg}(w,b)))$$

by the identity path, which is valid because we have a judgmental equality

$$\begin{aligned}
&\Phi((h,\beta))'(w,b)\\
&\quad\equiv (\Phi_1((h,\beta),w) \circ \mathsf{subtree\_addr}(b),\\
&\qquad \Phi_2((h,\beta),w) \circ \mathsf{subtree\_addr}(b))\\
&\quad\equiv \Phi((h,\beta), \mathsf{arg}(w,b)).
\end{aligned}$$

For the last equality we use that $\kappa(\lambda b.\,\mathsf{refl})$ is equal to to

$$\begin{aligned}
&\kappa(\lambda b.\,\mathsf{refl})\\
&\quad\equiv \mathsf{ap}_{s_C(\mathsf{label}(w),-)}(\mathsf{funext}(\lambda b.\,\mathsf{ap}_{\mathsf{pr}_1(-)(\mathsf{root\_addr})}\mathsf{refl}))\\
&\quad\equiv \mathsf{ap}_{s_C(\mathsf{label}(w),-)}(\mathsf{funext}(\lambda b.\,\mathsf{refl}))\\
&\quad= \mathsf{ap}_{s_C(\mathsf{label}(w),-)}(\mathsf{refl})\\
&\quad\equiv \mathsf{refl}.
\end{aligned}$$

□

Now we can combine those lemmas to an equivalence

Lemma 5.1.20.

$$\mathsf{WHom}' \simeq \prod_{w} \mathsf{LHom}(w).$$

Proof. By Lemmas 5.1.17 and 5.1.19 the function Φ is an equivalence from WHom' to $\prod_{(w)} \mathsf{LHom}(w)$. $\square$

Corollary 5.1.21. *The type* $\mathsf{WHom}((\mathsf{W}', \mathsf{sup}'), (C, s_C))$ *is contractible.*

Proof. By the equivalence

$$\mathsf{WHom}((\mathsf{W}', \mathsf{sup}'), (C, s_C)) \simeq \mathsf{WHom}' \simeq \prod_{w} \mathsf{LHom}(w)$$

from Lemmas 5.1.12 and 5.1.20 and contractibility of $\prod_{(w)} \mathsf{LHom}(w)$ from Corollary 5.1.10. $\square$

Lemma 5.1.22. *The algebra* $(\mathsf{W}', \mathsf{sup}')$ *is initial.*

Proof. Initiality means contractibility of the type of homomorphisms from $(\mathsf{W}', \mathsf{sup}')$ to every other $(\!|A \blacktriangleright B|\!)$-algebra. Because the algebra (C, s_C) was fixed arbitrarily, this follows from Corollary 5.1.21. $\square$

Theorem 5.1.23. *For evey type A and family $B : A \to \mathcal{U}$, there is an initial algebra of the functor $(\!|A \blacktriangleright B|\!)$.*

Proof. By the construction of $(\mathsf{W}', \mathsf{sup}')$ and Lemma 5.1.22. $\square$

5.2 Recursion with Judgmental Computation Rule

Similar to our construction of M-types we can use propositional resizing to refine W' and get a judgmental computation rule. However, our construction only archieves this for the simple non-dependent recursor which is equivalent to the family of algebra morphism from W'. The computation rule for the more general induction principle still holds only as a prospositional equality. Like with M-types this is not specific to W' and works on every initial algebra for the functor $[\![A \rhd B]\!]$.

For the definition of W we use a similar idea as for the definition of W'.

Definition 5.2.1. An element of W consists of

- an element of $w' : \mathsf{W}'$,
- a function r that is intended to mimic the application of the recursor on w',
- a proof that r actually agrees with the recursor.

More precisely, we define the type W as

$$\mathsf{W} :\equiv \sum_{w':\mathsf{W}'} \textstyle\sum_{(r:\sum_{(P:\mathcal{U})}(\sum_{(a)} B(a) \to P) \to P)} (\lambda P.\, \lambda s_P.\, \mathsf{rec}_{\mathsf{W}'}(P, s_P, w')) = r.$$

Now we do not need the recursor on W' anymore, but can apply our own version instead.

Definition 5.2.2. We define the **recursor** on W as

$$\mathsf{rec}_{\mathsf{W}}(P, s_P, (w', r, p)) :\equiv r(P, s_P).$$

For the definition of the constructor we have two options:

(i) Apply the recursor on W' as the second component,
(ii) Use the second component of all subtrees to build a new recursor.

Of course only the second option gives us any benefit over W'.

Definition 5.2.3. We define the **constructor** on W component-wise with the first two components being

$$\begin{aligned}
\mathsf{sup}_1(a, f) &:\equiv \mathsf{sup}'(a, \mathsf{pr}_1 \circ f) \\
\mathsf{sup}_2(a, f) &:\equiv \lambda P.\, \lambda s_P.\, s_P(a, \lambda b.\, \mathsf{pr}_1(\mathsf{pr}_2(f(b)))(P, s_P)).
\end{aligned}$$

We define the third component $\mathsf{sup}_3(a, f)$ as the proof

$$\begin{aligned}
&\lambda P.\, \lambda s_P.\, \mathsf{rec}_{\mathsf{W}'}(P, s_P, \mathsf{sup}_1(a, f)) \\
&\quad = \lambda P.\, \lambda s_P.\, s_P(a, \mathsf{rec}_{\mathsf{W}'}(P, s_P) \circ \mathsf{pr}_1 \circ f) \\
&\quad \equiv \lambda P.\, \lambda s_P.\, s_P(a, \lambda b.\, (\lambda P.\, \lambda s_P.\, \mathsf{rec}_{\mathsf{W}'}(P, s_P, \mathsf{pr}_1(f(b))))(P, s_P)) \\
&\quad = \lambda P.\, \lambda s_P.\, s_P(a, \lambda b.\, \mathsf{pr}_1(\mathsf{pr}_2(f(b)))(P, s_P)) \\
&\quad \equiv \mathsf{sup}_2(a, f).
\end{aligned}$$

The first step is an application of the computation rule for $\mathsf{rec}_{\mathsf{W}'}$. In the second step we apply $\mathsf{pr}_2(\mathsf{pr}_2(f(b)))$.

Finally we combine all components to

$$\mathsf{sup}(a, f) :\equiv \Big(\mathsf{sup}_1(a, f), \big(\mathsf{sup}_2(a, f), \mathsf{sup}_3(a, f)\big)\Big).$$

Remark 5.2.4. As promised, these definitions satisfy the *computation rule* judgmentally:

$$\begin{aligned}
&\mathsf{rec}_{\mathsf{W}}(P, s_P, \mathsf{sup}(a, f)) \\
&\quad \equiv \mathsf{rec}_{\mathsf{W}}(P, s_P, (\dots, \lambda P.\, \lambda s_P.\, s_P(a, \lambda b.\, \mathsf{pr}_1(\mathsf{pr}_2(f(b)))(P, s_P), \dots)) \\
&\quad \equiv s_P(a, \lambda b.\, \mathsf{pr}_1(\mathsf{pr}_2(f(b)))(P, s_P) \\
&\quad \equiv s_P(a, \mathsf{rec}_{\mathsf{W}}(P, s_P) \circ f).
\end{aligned}$$

The only thing that is left to show, is the initiality of our newly defined algebra $(\mathsf{W}, s_{\mathsf{W}})$, where $s_{\mathsf{W}}((a, f)) :\equiv \mathsf{sup}(a, f)$ defines the uncurried version of our constructor. We already know that $(\mathsf{W}', s_{\mathsf{W}'})$ is initial, so it suffices to show $(\mathsf{W}, s_{\mathsf{W}}) = (\mathsf{W}', s_{\mathsf{W}'})$. We start with an equivalence between both carriers:

Lemma 5.2.5.

$$\mathsf{W} \simeq \mathsf{W}'.$$

Proof. We define functions g and h in both directions and show that h is quasi-inverse to g:

$$\begin{aligned} g((w', (-,-))) &:\equiv w' \\ h(w') &:\equiv (w', (\lambda P.\, \lambda s_P.\, \mathsf{rec}_{\mathsf{W}'}(P, s_P, w'), \mathsf{refl})). \end{aligned}$$

The function h is a right inverse by judgmental equality. To prove that it is a left inverse, fix $(w', (r,p)) : \mathsf{W}$. We show

$$\begin{aligned} &h(g((w', (r,p)))) \\ &\quad \equiv (w', (\lambda P.\, \lambda s_P.\, \mathsf{rec}_{\mathsf{W}'}(P, s_P, w'), \mathsf{refl})) \\ &\quad = (w', (r,p)) \end{aligned}$$

by based path induction on $p : (\lambda P.\, \lambda s_P.\, \mathsf{rec}_{\mathsf{W}'}(P, s_P, w')) = r$. □

Lemma 5.2.6.

$$(\mathsf{W}, s_{\mathsf{W}}) = (\mathsf{W}', s'_{\mathsf{W}}).$$

Proof. Let g and h be the two functions from the proof of Lemma 5.2.5 and e the resulting equivalence. Equality on the first component follows from e by univalence. Equality on the second component holds by

$$\begin{aligned} &(\mathsf{ua}(e)_*(s_{\mathsf{W}}))((a,f)) \\ &\quad = \mathsf{ua}(e)_*\Big(\mathsf{sup}(a, \mathsf{ua}(e)^{-1}{}_*(f))\Big) \\ &\quad = g(\mathsf{sup}(a, h \circ f)) \\ &\quad \equiv g((\mathsf{sup}'(a, \mathsf{pr}_1 \circ h \circ f), \ldots, \ldots)) \\ &\quad \equiv g((\mathsf{sup}'(a, f), \ldots, \ldots)) \\ &\quad \equiv s'_{\mathsf{W}}((a,f)). \end{aligned}$$

In the first step we apply Fact 2.4.5 and in step three we apply Fact 2.6.4. □

Theorem 5.2.7. *The algebra* $(\mathsf{W}, s_{\mathsf{W}})$ *is initial.*

Proof. By initiality of $(\mathsf{W}', s_{\mathsf{W}'})$ and Lemma 5.2.6. □

Theorem 5.2.8. *For evey type A and family $B : A \to \mathcal{U}$, there is an initial algebra $(\mathsf{W}_{(a:A)}B(a), s)$ of the functor $(\!| A \blacktriangleright B |\!)$ It has a recursor*

$$\mathsf{rec}_{\mathsf{W}_{(a:A)}B(a)} : \prod_{C:\mathcal{U}} \left(\prod_{a:A} (B(a) \to C) \to C \right) \to \mathsf{W}_{(a:A)}B(a) \to C$$

which satisfies the computation rule

$$\mathsf{rec}_{\mathsf{W}_{(a:A)}B(a)}(C, g, s(a,f)) \equiv s(a, \mathsf{rec}_{\mathsf{W}_{(a:A)}B(a)}(C, g) \circ f).$$

Proof. By the construction of $(\mathsf{W}, \mathsf{sup})$, Theorem 5.2.7 and Remark 5.2.4. □

6 Construction of Basic Types

In this chapter we show that we can drop some of the most basic type constructs from our core theory and replace them by definitions with the exact same behavior, including judgmental equalities. This justifies the use of the common pattern-matching syntax on types after we redefined them. We will define the empty type $\mathbf{0}$, the unit type $\mathbf{1}$, the type of booleans $\mathbf{2}$, coproduct $A + B$ of types A and B, the option type $\mathsf{Option}(A)$ and the propositional truncation $\|A\|$ of a type A. The order is important because we need $\mathbf{0}$ and $\mathbf{1}$ for the definition of $\mathbf{2}$ and $\mathbf{2}$ for the definition of coproducts. It is not surprising that those definitions are possible in our theory, but it is notable that we get the desired judgmental equalities.

6.1 The Empty Type

The empty type has no constructors and is characterized only by its induction eliminator $\mathsf{rec}_{\mathbf{0}} : \prod_{(C:\mathcal{U})} \mathbf{0} \to C$. We define the empty type as

$$\mathbf{0} :\equiv 0 = 1.$$

For the eliminator assume that we have $C : \mathcal{U}$ and an inhabitant $x : \mathbf{0}$. We will use p for a transport in a family of types $P : \mathbb{N} \to \mathcal{U}$, such that $P(0)$ is some inhabited type and $P(1)$ is C.

$$\begin{aligned} P(0) &:\equiv \mathbb{N} \\ P(\mathsf{succ}(x)) &:\equiv C \end{aligned}$$

$$\mathsf{rec}_{\mathbf{0}}(C, x) :\equiv \mathsf{transport}^{P}(x, 0)$$

We don't need to prove any computation rules, because there are no elements in $\mathbf{0}$.

6.2 The Unit Type

We define the unit type and its constructor as

$$\begin{aligned} \mathbf{1} &:\equiv \sum_{n:\mathbb{N}} 0 = n \\ \star &:\equiv (0, \mathsf{refl}_0). \end{aligned}$$

The induction principle $\mathsf{ind}_{\mathbf{1}} : \prod_{(C:\mathbf{1}\to\mathcal{U})} C(\star) \to \prod_{(x:\mathbf{1})} C(x)$ is defined by a based path induction. Fix a family $C : \mathbf{1} \to \mathcal{U}$, an element $c : C(\star)$ and a pair $(n, p) : \mathbf{1}$ as arguments.

With based path induction it suffices to consider the case where n is 0 and p is refl_0, hence (n,p) is $\star$. In this case we define

$$\mathsf{ind}_{\mathbf{1}}(C,c,(0,\mathsf{refl}_0)) :\equiv c.$$

The computation rule $\mathsf{ind}_{\mathbf{1}}(C,c,\star) = c$ holds by definition.

6.3 The Type of Booleans

We define the type of booleans as the subtype of natural numbers that are less than 2.

$$\mathbf{2} :\equiv \sum_{n:\mathbb{N}} n < 2.$$

The *less* relation is defined recursively with $\mathbf{0}$ and $\mathbf{1}$ as base cases.

$$\begin{aligned} n < 0 &:\equiv \mathbf{0} \\ 0 < \mathsf{succ}(n) &:\equiv \mathbf{1} \\ \mathsf{succ}(m) < \mathsf{succ}(n) &:\equiv m < n. \end{aligned}$$

The constructors just return the numbers 0 and 1 with corresponding proofs.

$$\begin{aligned} 0_{\mathbf{2}} &:\equiv (0,\star) \\ 1_{\mathbf{2}} &:\equiv (1,\star) \end{aligned}$$

For the definition of the induction principle $\mathsf{ind}_{\mathbf{2}} : \prod_{(C:\mathbf{2}\to\mathcal{U})} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \prod_{(x:\mathbf{2})} C(x)$ we will use pattern-matching on the natural number and exclude all cases for numbers of the form $\mathsf{succ}(\mathsf{succ}(n))$ by elimination on the proof of $\mathsf{succ}(\mathsf{succ}(n)) < 2 \equiv \mathbf{0}$.

$$\begin{aligned} \mathsf{ind}_{\mathbf{2}}(C,c_0,c_1,(0,\_)) &:\equiv c_0 \\ \mathsf{ind}_{\mathbf{2}}(C,c_0,c_1,(1,\_)) &:\equiv c_1. \end{aligned}$$

The computation rules coincide with the defining equations.

6.4 Coproducts

Fix types A and B. Elements of $A+B$ will be pairs consisting of a boolean in the first component, that tells us, if we have an element of the left or the right side. Depending on the first component, the second component will contain an element of either A or B.

$$\begin{aligned} A+B &:\equiv \sum_{b:\mathbf{2}} \mathrm{if}\, b\, \mathrm{then}\, A\, \mathrm{else}\, B \\ \mathsf{inl}(a) &:\equiv (0_{\mathbf{2}},a) \\ \mathsf{inr}(b) &:\equiv (1_{\mathbf{2}},b) \end{aligned}$$

The induction principle does pattern matching on the boolean.

$$\mathsf{ind}_{A+B} : \sum_{C:A+B\to\mathcal{U}} \left(\prod_{a:A} C(\mathsf{inl}(a))\right) \to \left(\prod_{b:B} C(\mathsf{inr}(b))\right) \to \prod_{x:A+B} C(x)$$

$$\begin{aligned} \mathsf{ind}_{A+B}(C, g_0, g_1, (0_{\mathbf{2}}, a)) &:\equiv g_0(a) \\ \mathsf{ind}_{A+B}(C, g_0, g_1, (1_{\mathbf{2}}, b)) &:\equiv g_1(b). \end{aligned}$$

The computation rules hold by definition.

6.5 Options

Given a type A, we define the option and its constructors by

$$\mathsf{Option}(A) :\equiv A + 1$$

$$\begin{aligned} \mathsf{some}(a) &:\equiv \mathsf{inl}(a) \\ \mathsf{none} &:\equiv \mathsf{inr}(\star) \end{aligned}$$

and the induction principle by

$$\mathsf{ind}_{\mathsf{Option}(A)} : \prod_{C:\mathsf{Option}(A)\to\mathcal{U}} \left(\prod_{a:A} C(\mathsf{some}(a))\right) \to C(\mathsf{none}) \to \prod_{x:\mathsf{Option}(A)} C(x)$$

$$\begin{aligned} \mathsf{ind}_{\mathsf{Option}(A)}(C, c_{\mathsf{some}}, c_{\mathsf{none}}, \mathsf{inl}(a)) &:\equiv c_{\mathsf{some}}(a) \\ \mathsf{ind}_{\mathsf{Option}(A)}(C, c_{\mathsf{some}}, c_{\mathsf{none}}, \mathsf{inr}(\star)) &:\equiv c_{\mathsf{none}} \end{aligned}$$

The computation rule holds judgmentally.

6.6 Propositional Truncation

Given some type A, we define the propositional truncation impredicatively as

$$\|A\| :\equiv \prod_{P:\mathsf{Prop}} (A \to P) \to P.$$

This is a mere proposition because its codomain is always a mere proposition.

$$\begin{aligned} |a| &:\equiv \lambda(P : \mathsf{Prop}).\, \lambda(f : A \to P).\, f(a) \\ \mathsf{rec}_{\|A\|} &: \textstyle\prod_{(P:\mathsf{Prop})} (A \to P) \to \|A\| \to P \\ \mathsf{rec}_{\|A\|}(P, f, x) &:\equiv x(P, f) \end{aligned}$$

For this construction to work, it is necessary that we have propositional resizing, because the argument P can have any universe level, but x can only be applied to propositions with a lower universe level than its own. With propositional resizing this doesn't matter, since P is a mere proposition and hence inhabits the lowest universe. The computation rule holds judgmentally:

$$\begin{aligned}\mathsf{rec}_{\|A\|}(P, f, |a|) &\equiv (\lambda P.\, \lambda f.\, f(a))(P, f) \\ &\equiv f(a).\end{aligned}$$

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[2] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in Homotopy Type Theory. *arXiv preprint arXiv:1504.02949*, 2015.

[3] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25:e5, 2015.

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