Inductive-inductive definitions.

Forsberg, Fredrik Nordvall

Inductive-inductive definitions

Fredrik Nordvall Forsberg

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

2013

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)

Date 18/11/2013

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date 18/11/2013

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)

Date 18/11/2013

Abstract

The principle of inductive-inductive definitions is a principle for defining data types in Martin-Löf Type Theory. It allows the definition of a set A, simultaneously defined with a family $B : A \to \mathrm{Set}$ indexed over A. Such forms of definitions have been used by several authors in order to for example define the syntax of Type Theory in Type Theory itself. This thesis gives a theoretical justification for their use.

We start by giving a finite axiomatisation of a type theory with inductive-inductive definitions in the style of Dybjer and Setzer's axiomatisation of inductive-recursive definitions. We then give a categorical characterisation of inductive-inductive definitions as initial objects in a certain category. This is presented using a general framework for elimination rules based on the concept of a Category with Families. To show consistency of inductive-inductive definitions, a set-theoretical model is constructed. Furthermore, we give a translation of our theory with a simplified form of the elimination rule into the already existing theory of indexed inductive definitions. This translation does not seem possible for the general elimination rule. Extensions to the theory are investigated, such as a combined theory of inductive-inductive-recursive definitions, more general forms of indexing and arbitrarily high (finite) towers of inductive-inductive definitions. Even so, not all uses of inductive-inductive definitions in the literature (in particular the syntax of Type Theory) are covered by the theories presented. Finally, two larger, novel case studies of the use of inductive-inductive definitions are presented: Conway's Surreal numbers and a formalisation of positive inductive-recursive definitions.

Table of Contents

Acknowledgements

I would like to thank my supervisor Anton Setzer: for everything I have learnt during my PhD, for always encouraging me, and for suggesting the topic of this thesis. Many thanks also to my other coauthors: Thorsten Altenkirch, Peter Morris, Neil Ghani, Lorenzo Malatesta, Helmut Schwichtenberg and Kenji Miyamoto. I have enjoyed working with all of you.

I would like to thank my examiners Peter Dybjer and John Tucker for their detailed feedback on this dissertation, as well as interesting discussions, and not only during the viva. This thesis has further benefited from discussions with Andreas Abel, Stevan Andjelkovic, Ulrich Berger, Nils Anders Danielsson, Clement Fumex, Marcelo Fiore, Peter Hancock, Paul Levy, Conor McBride, and Monika Seisenberger, and of course my collaborators listed above.

Liam O'Reilly, Pierre-Evariste Dagand and Angharad Williams proof read parts of this thesis, for which I am grateful.

I would further like to thank all current and former members of the Theoretical Computer Science group at Swansea University for such a stimulating working environment. In the department in general, I am especially thankful to Matt Gwynne, Phil and Emma James, Liam O'Reilly, Tom Owen and Jen Pearson for making me feel welcome in Wales. I am equally grateful to Helmut Schwichtenberg and the Logic group at LMU Munich for hosting me there for five months.

I have received support from the Engineering and Physical Sciences Research Council through grant EP/G033374/1, *Theory And Applications of Induction Recursion*, as well as from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement nº 238381.

CHAPTER 1

Introduction

Contents

This thesis describes a novel class of data types in the context of Martin-Löf Type theory. We argue that expressive data types are important, both from a foundational point of view (using Type Theory as a foundation for constructive mathematics) and for the programmer that wishes to write programs that are correct by construction.

This chapter provides an introduction to the main body of the thesis. We first emphasise the importance of expressive data types. We then review the history of inductive definitions – both inside and outside of Type Theory – before giving an overview of the rest of the thesis and the publications it is based upon.

1.1 The importance of dependent types

We start by motivating the use of dependent types for programming and mathematics.

1.1.1 Data types for computer science

Software is becoming increasingly important and widespread in modern society. It is also becoming increasingly complex. At the same time, with computer programs appearing in more and more everyday and safety-critical devices, the cost of failure is increasing as well. How can we effectively develop programs, and be sure that they are correct?

One option for the programmer is to tell the computer more of their intentions, so that it can help spot errors or even derive trivial parts of programs. The question then becomes how we can tell the computer what we want. There are of course many options, for example formal specifications, exhaustive testing frameworks or refinement development, to name a few.

This thesis pursues another solution, which begins with a simple observation: even the earliest programming languages such as FORTRAN [IBM Applied Science Division, 1954] and ALGOL [Perlis and Samelson, 1958] had a facility for the programmer to tell the compiler some of her intentions – a type system[1]. For example, consider the following simple Haskell [Marlow, 2010] program snippet:

$$\begin{aligned} &\mathsf{f} : \mathsf{Integer} \to \mathsf{Integer} \\ &\mathsf{f}\, x = \mathsf{True} \end{aligned}$$

The programmer is telling the compiler that she intends to write a function f which takes an integer x as input and returns an integer. However, both the compiler and the reader can easily spot the error the programmer has made: she is not returning an integer after all, but is trying to return the Boolean True! Luckily, a Haskell compiler would be clever enough to inform our programmer of her mistake.

But now consider the following program:

$$\begin{aligned} &\mathsf{sort} : \mathsf{List\ Integer} \to \mathsf{List\ Integer} \\ &\mathsf{sort}\, xs = [\,2, 0, 3\,] \end{aligned}$$

This time, the compiler does not complain. The reader, however, should! It is not such a wild guess that the programmer intended for this function to return a sorted permutation of its input. This is exactly the kind of information we would like to pass on to the compiler. As it stands, though, the input is irrelevant and the output is not even sorted. If we want the compiler to alert us to this fact, we should make the type system more expressive. Martin-Löf Type Theory [Martin-Löf, 1972, 1984; Nordström et al., 1990, 2001] is a programming language with such an expressive type system. Even though intended as a foundational system for constructive mathematics, Martin-Löf [1982] has also stressed the connection to programming. The result is very expressive. By exploiting the Curry-Howard isomorphism (see Section 2.1.5), we can encode any specification written as a first-order formula in the type system.

First-order logic in itself is not enough. Without meaningful atomic propositions, we cannot specify many properties. By the Curry-Howard isomorphism again, atomic propositions correspond to ground types in Type Theory[2]. Thus, it is important to extend Type Theory with a large collection of basic data types as well. By the dual logical and mathematical nature of Type Theory, these will be used both for computation and for reasoning.

We will review current systems of data types for Type Theory in Section 1.2. This thesis studies a class of data types, called ***inductive-inductive definitions*** for reasons that

[1] Of course, these early type systems were not *introduced* for this purpose, but were rather meant for the programmer to help the compiler, not the other way around (different data types require different memory layouts). Likewise, type theory does not begin with FORTRAN, but rather with Russel [1903, Appendix B].

[2] From now on, we will simply write Type Theory for Martin-Löf Type Theory.

will become clear below, which generalises most of the data types previously considered. With inductive-inductive definitions, we can give sort the type

$$\mathsf{sort} : \mathsf{List\ Integer} \to \mathsf{SortedList\ (Integer}, \leq)$$

which would rule out the implementation above – the list [2,0,3] is not sorted. See Example 3.2 for the inductive-inductive definition of a data type of sorted lists.

We could also consider to go further. The type given to sort above does not guarantee the correctness of the function; a possible function of that type would for instance be

$$\begin{aligned}&\mathsf{wrong\text{-}sort} : \mathsf{List\ Integer} \to \mathsf{SortedList\ (Integer}, \leq)\\&\mathsf{wrong\text{-}sort}\ xs = []\end{aligned}$$

since the empty list [] certainly is sorted. To be sure that we have a correct program, we could give sort the dependent[3] type

$$\mathsf{sort} : (xs : \mathsf{List\ Integer}) \to (\Sigma\ ys : \mathsf{SortedList\ (Integer}, \leq))\ (\mathsf{Permutation}\ xs\ ys)$$

where Permutation xs ys is a data type consisting of proofs that xs is a permutation of ys. Such a data type can also be defined using inductive-inductive definitions (in fact, indexed inductive definitions, see Appendix A.1.2).

It is important to point out that types are not only there to tell us about our mistakes after we have made them. Instead, types can also guide us towards the program we would like to write, or even help the compiler to automatically derive parts of the program for us. For this, dependent types are crucial; if we ask for a list, any list, we might be disappointed by the result – most lists are not sorted! However, if we ask for a sorted list that is a permutation of the input, we will be much happier with whatever the compiler is coming up with for us.

1.1.2 Data types for the working mathematician

So far, we have focused on the needs of the programmer. But, as Martin-Löf [1982] points out:

> If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute [. . .], then it no longer seems possible to distinguish the discipline of programming from constructive mathematics.

Hence data types should be important also for the constructive mathematician. For constructive mathematics in the style of Bishop [1967], where mathematics is done informally (but without the use of the principle of excluded middle), this is perhaps not immediately obvious. Martin Löf Type Theory was invented with the goal of being "a full-scale system for intuitionistic mathematics" [Martin-Löf, 1972] carried out in such

[3] See Section 2.1 for notation and background on dependent types.

an informal fashion. Thus, if a mathematical object has been informally constructed, it is important that the underlying formal system is able to faithfully represent this object as well. We will see examples of how inductive-inductive definitions support such informal mathematical developments in Chapter 7.

1.2 Inductive definitions in Type Theory and set theory

Inductive definitions are ubiquitous in mathematics, perhaps especially so in constructive circles. In this section, we review the basic approaches to formal systems including such definitions, both in Type Theory, set theory and first order logic.

1.2.1 Inductive definitions in Type Theory

Martin-Löf's formulations of Type Theory [Martin-Löf, 1972, 1982, 1984] includes *inductive* definitions of, for example, disjoint unions $A + B$, the identity set $x \equiv_A y$, finite sets $\mathsf{Fin}(n)$, the natural numbers $\mathbb{N}$, well-orderings $\mathsf{W}(x : A)B(x)$ and lists List_A, as well as an *inductive-recursive* definition of a universe (U, T) à la Tarski; we will come back to these specific type formers as examples of classes of data types in Section 1.2.1.3. It is understood that further data types may be added, as long as they are meaningful, i.e. are supported by the semantics of the language.

This however raises the question: what extensions are meaningful? The possibility of developing a general formulation of meaningful extensions was mentioned already by Martin-Löf [1982][4]:

> The type $\mathbb{N}$ is just the prime example of a type introduced by an *ordinary inductive definition*. However, it seems preferable to treat this special case rather than to give a necessarily much more complicated general formulation which would include $(\Sigma x : A)B(x)$, $A + B$, $\mathsf{Fin}(n)$ and $\mathbb{N}$ as special cases. See Martin-Löf [1971] for a general formulation of inductive definitions in the language of ordinary first order predicate logic.

1.2.1.1 Schemata of inductive definitions

An early schema of inductive definitions is presented by Martin-Löf [1971], as referred to in the quote above. In fact, it is so early that it predates Type Theory, and is instead couched in first-order logic. Backhouse [1988] (see also Backhouse et al. [1989]) was the first to give a general formulation of "disciplined extensions" of Type Theory. By defining a schema of inductive definitions, Backhouse shows how the elimination and computation rules can be automatically derived from the introduction rules, which can considerably simplify the presentation of the theory. The idea that the elimination rules for first-order logic are derivable can be found in Prawitz [1979]. In computer science, the same idea can implicitly be found in Burstall [1969], where (simply typed)

[4] The notation for the different data types in the quoted text has been changed to coincide with the rest of this thesis.

data types such as lists and trees are given by constructors, i.e. only their introduction rules are specified.

However, the schema given by Backhouse allowed inconsistent definitions, since it did not enforce strict positivity (this is also remarked upon in the conclusion of the article). Dybjer [1994] gave a different schema, which only gives rise to consistent definitions. Dybjer proves this using a set-theoretic semantics [Dybjer, 1991]. The schema also extends Backhouse's schema in several ways, most notably by also allowing the inductive definition of a family of types, i.e. an *indexed* inductive definition. Coquand and Paulin-Mohring [1990] give a similar schema in the setting of the Calculus of Constructions [Coquand and Gérard, 1988], i.e. impredicative Type Theory.

There are other constructively justified forms of induction-like definitions, such as Martin-Löf's universe of small types, that are not covered by the schemas discussed so far. Dybjer [2000] proposed another schema of inductive-recursive definitions, which does cover Martin-Löf's universe and many other examples, such as Martin-Löf's computability predicates [Martin-Löf, 1972] or Aczel's Frege structures [Aczel, 1980]. Setzer was interested in inductive-recursive definitions as a proof-theoretically strong extension of Type Theory, but found the schematic presentation too imprecise for proof-theoretical analysis. To remedy this, he developed a finite axiomatisation together with Dybjer [Dybjer and Setzer, 1999] by internalising the schema. This axiomatisation was then further studied and extended [Dybjer and Setzer, 2003, 2006]. We take much inspiration from their work in the current thesis. The idea of representing data types internally in Type Theory has been used for generic programming [Benke et al., 2003; Morris, 2007; Magalhães, 2012], and forms the basis for all data types in Epigram 2 [Chapman et al., 2010].

1.2.1.2 Other approaches to inductive definitions

We will use an internalised schema of data type definitions in this thesis. Nonetheless, let us discuss some other approaches to inductive definitions, and why they are not suitable for intensional Type Theory.

Containers [Abbott et al., 2005] and indexed containers [Altenkirch and Morris, 2009] give a more semantic view of data types, without e.g. syntactical criterions of strict positivity. This is very similar to representing inductive definitions by W-types [Dybjer, 1997; Abbott et al., 2004], but makes essential use of extensional Type Theory, which we wish to avoid for the initial axiomatisation because of its not so nice meta-theoretical properties (see Section 2.1.6.1). We will explore an inductive-inductive extension of containers in Section 5.2.

Another option is to use an impredicative Church or Scott encoding of data types. Pfenning and Paulin-Mohring [1990] explore such encodings for the Calculus of Constructions, but this is not a possible solution in Martin-Löf Type Theory, which is a predicative theory. Furthermore, Church encodings only give rise to non-dependent elimination principles [Geuvers, 2001].

Yet another option is to add a (least) fixed point operator to the theory [Mendler, 1987]. This is not so different from the schema approach; for instance, one still has to make sure that the fixed point operator is only applied to strictly positive expressions. Conceptually, the approach is not so clear, however, as inductive types are represented as equirecursive types, and hence e.g. the natural numbers $\mathbb{N} = \mu X.(1 + X)$ are both a sum type and not a sum type at the same time; this often requires the use of subset types in order to make sense.

1.2.1.3 Different classes of data types in Type Theory

Let us now look at some examples of inductive definitions, such as the natural numbers, lists, well-orderings, the identity set, finite sets, and a universe à la Tarski. These examples can be categorised as different kinds of inductive definitions.

The first few (up to well-orderings) are just ordinary inductive definitions, where a single set is defined inductively. A typical example is the type $\mathsf{W}(A, B)$ of well-orderings, parameterised by $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$. The introduction rule is:

$$\frac{a : A \qquad f : B(a) \to \mathsf{W}(A, B)}{\mathsf{sup}(a, f) : \mathsf{W}(A, B)}$$

Each element of $\mathsf{W}(A, B)$ can be thought of as a well-founded tree, where the set A contains the possible branching types of the tree and $B : A \to \mathsf{Set}$ describes the branching degree of each type. Thus $\mathsf{sup}(a, f) : \mathsf{W}(A, B)$ is a tree with top-most branching type $a : A$, "above" all the subtrees $f(b)$ for $b : B(a)$ (hence the constructor name 'sup'). Here $a : A$ is a *non-inductive* argument, whereas $f : B(a) \to \mathsf{W}(A, B)$ is an *inductive* argument because of the occurrence of $\mathsf{W}(A, B)$. Note how the later argument depends on the earlier non-inductive argument.

The identity type and the finite sets are examples of *inductive families*, where a whole family $X : I \to \mathsf{Set}$ is defined inductively at the same time, for some fixed index set I. For the family $\mathsf{Fin} : \mathbb{N} \to \mathsf{Set}$ of finite sets (i.e. $\mathsf{Fin}(n)$ is a set with n elements), the index set is the natural numbers $\mathbb{N}$. We have introduction rules

$$\frac{n : \mathbb{N}}{\mathsf{z}_n : \mathsf{Fin}(n + 1)} \qquad \frac{n : \mathbb{N} \qquad m : \mathsf{Fin}(n)}{\mathsf{s}_n(m) : \mathsf{Fin}(n + 1)}$$

Thus, indeed, the type $\mathsf{Fin}(n + 1)$ has $n + 1$ elements, which can be enumerated as z_n, $\mathsf{s}_n(\mathsf{z}_{n-1})$, $\mathsf{s}_n(\mathsf{s}_{n-1}(\mathsf{z}_{n-2}))$ up to $\mathsf{s}_n(\mathsf{s}_{n-1}(\cdots \mathsf{s}_1(\mathsf{z}_0)))$. The type of the inductive argument $m : \mathsf{Fin}(n)$ of the second rule has index n, which is different from the index $n + 1$ of the type of the constructed element. Thus the whole family has to be defined at the same time.

The universe à la Tarski is an example of an *inductive-recursive definition*, where a set U is defined inductively together with a recursive function $T : U \to \mathsf{Set}$. The constructors for U may depend negatively on T applied to elements of U, as is the case if U, for example, is closed under dependent function spaces:

$$\frac{a : U \qquad b : T(a) \to U}{\pi(a, b) : U}$$

with $T(\pi(a,b)) = (x : T(a)) \to T(b(x))$.

In the last example, $T : U \to \mathsf{Set}$ was defined recursively. Sometimes, however, one might not want to give $T(u)$ completely as soon as $u : U$ is introduced, but instead define T inductively as well. This is the principle of *inductive-inductive definitions.* A set A is inductively defined simultaneously with an A-indexed set B, which is also inductively defined, and the introduction rules for A may also refer to B. Typical introduction rules might take the form

$$\frac{a : A \quad b : B(a) \quad \ldots}{\mathsf{intro}_A(a, b, \ldots) : A} \qquad \frac{a_0 : A \qquad b : B(a_0) \quad a_1 : A \quad \ldots}{\mathsf{intro}_B(a_0, b, a_1, \ldots) : B(a_1)}$$

Notice that this is not a simple mutual inductive definition of two sets, as B is indexed by A. It is not an ordinary inductive family either, as A may refer to B. Finally, it is not an instance of induction-recursion, as B is constructed inductively, not recursively.

1.2.1.4 Inductive definitions versus recursive definitions

In both an inductive-inductive and an inductive-recursive definition, a set U and a family $T : U \to \mathsf{Set}$ are defined simultaneously. The difference between the two principles is how T is defined: inductively or recursively. In the following, we first discuss the difference between an inductive and a recursive definition. To exemplify this difference, consider the following two definitions of a data type $\mathsf{Nonempty} : \mathbb{N} \to \mathsf{Set}$ of non-empty lists of a certain length (with elements from a set A):

Inductive definition The singleton list $[a]$ has length 1; and if a is an element, and the list ℓ has length n, then $\mathsf{cons}(a, \ell)$ is a list of length $n + 1$. As an inductive definition, this becomes

$$\frac{a : A}{[a] : \mathsf{Nonempty}_{\mathrm{ind}}(1)} \qquad \frac{a : A \qquad \ell : \mathsf{Nonempty}_{\mathrm{ind}}(n)}{\mathsf{cons}(a, \ell) : \mathsf{Nonempty}_{\mathrm{ind}}(n + 1)}$$

Notice that there is no constructor which constructs elements of length 0, i.e. in the set $\mathsf{Nonempty}_{\mathrm{ind}}(0)$.

Recursive definition In the recursive definition of the data type, we define the set $\mathsf{Nonempty}_{\mathrm{rec}}(n)$ for every natural number:

$$\begin{aligned} \mathsf{Nonempty}_{\mathrm{rec}}(0) &= \mathbf{0} \\ \mathsf{Nonempty}_{\mathrm{rec}}(1) &= A \\ \mathsf{Nonempty}_{\mathrm{rec}}(n + 2) &= A \times \mathsf{Nonempty}_{\mathrm{rec}}(n + 1) \end{aligned}$$

In the recursive definition, $\mathsf{Nonempty}_{\mathrm{rec}}(k)$ is defined in one go, whereas the inductively defined $\mathsf{Nonempty}_{\mathrm{ind}}(k)$ is built up from below. In order to prove that the set $\mathsf{Nonempty}_{\mathrm{ind}}(0)$ is empty, one has to carry out a proof by induction over $\mathsf{Nonempty}_{\mathrm{ind}}$.

This difference is now carried over to an inductive-recursive/inductive-inductive definition of $U : \mathsf{Set}$, $T : U \to \mathsf{Set}$. In an inductive-inductive definition, T is generated inductively, i.e. given by a constructor $\mathsf{intro}_T : (x : F(U,T)) \to T(i(x))$ for some (strictly positive) functor F. In an inductive-recursive definition, on the other hand, T is defined by recursion on the way the elements of U are generated. This means that $T(\mathsf{intro}_U(x))$ must be given completely as soon as the constructor $\mathsf{intro}_U : G(U,T) \to U$ is introduced.

There are some practical differences between the two approaches. An inductive-inductive definition gives more freedom to describe the data type, in the sense that many different constructors for T can contribute to the set $T(\mathsf{intro}_U(x))$. However, because of the inductive generation of T, T can only occur positively in the type of the constructors for U (and T), whereas T can occur also negatively in an inductive-recursive definition.

Finally, as long as $U : \mathrm{Set}$ is inductively defined, it makes sense to define $T : U \to D$ recursively for an arbitrary codomain D, such as e.g. $D = \mathrm{Set}$ or $D = \mathbb{N}$. By contrast, it does not make sense to define e.g. $T : U \to \mathbb{N}$ inductively: this would mean that $T(u) : \mathbb{N}$ should be given by constructors, which is nonsense (a natural number such as 17 does not have elements!). Thus, in an inductive-inductive definition, we are restricted to families $T : U \to \mathrm{Set}$ with codomain Set, since only then does it make sense to be given by constructors. We will see in Section 6.2 that the domain of T can be made more general for inductive-inductive definitions. See also Ghani et al. [2013b] for such extensions for inductive-recursive definitions.

1.2.2 Inductive definitions in set theory

We give a quick account of inductive definitions in (classical) set theory using Aczel's *rule sets*. For a more detailed exposition, the reader is referred to Aczel [1977], from which much of the following material is taken. We will use results from this section in Section 5.1.

Definition 1.1 Let A be a set.

(i) A *rule* on the base set A is a pair (X, x) where $X \subseteq A$ is a set – the set of *premises* – and $x \in A$ – the *conclusion* of the rule. We often write a rule as

$$\frac{X}{x} \ ,$$

reminiscent of natural deduction.

(ii) Let Φ be a set of rules on A (a *rule set* on A). A set $Y \subseteq A$ is *Φ-closed* if for each rule $\frac{X}{x} \in \Phi$, it is the case that $X \subseteq Y$ implies $x \in Y$, i.e. if the premises of a rule are contained in Y, then so is the conclusion.

(iii) Let Φ be a rule set. The least Φ-closed set is called *inductively defined by* Φ. ■

Using impredicativity, the set inductively defined by Φ can always be constructed as

$$\mathcal{I}(\Phi) := \bigcap\{Y \subseteq A \mid Y \text{ is } \Phi\text{-closed}\} \ .$$

This is an intersection over a non-empty set, as for example A itself is trivially Φ-closed. Furthermore, the intersection of any collection of Φ-closed sets is Φ-closed. Hence $\mathcal{I}(\Phi)$ indeed is the least Φ-closed set.

Examples 1.2 We give some examples of sets defined by rule sets.

(i) The natural numbers are perhaps the most familiar example of an inductively defined set. They are defined by the rule set (on $\mathbb{R}$, say)

$$\{\frac{\varnothing}{0}\} \cup \{\frac{\{n\}}{n+1} \mid n \in \mathbb{N}\} \ .$$

(ii) Let $\mathbb{C}$ be a category and let X be a set of morphisms from $\mathbb{C}$. There is a smallest subcategory $\mathbb{C}_X$ of $\mathbb{C}$ which contains the morphisms in X. The objects of $\mathbb{C}_X$ are the domains and codomains of the morphisms in X, and the morphisms are inductively defined by the rule set (on morphisms from $\mathbb{C}$)

$$\{\frac{\varnothing}{f} \mid f \in X\} \cup \{\frac{\varnothing}{\mathrm{id}_A} \mid \text{A object in } \mathbb{C}_X\} \cup \{\frac{\{f,g\}}{f \circ g} \mid f, g \text{ composable morphisms in } \mathbb{C}\} \ .$$

(iii) Well-formed arithmetical terms built up from constants 0, 1, a unary operator − and binary operators + and × are inductively defined by the rule set (on the set S of finite strings on the alphabet $\{0, 1, -, +, \times, (,)\}$)

$$\{\frac{\varnothing}{0}\} \cup \{\frac{\varnothing}{1}\} \cup \{\frac{\{x\}}{-(x)} \mid x \in S\} \cup \{\frac{\{x,y\}}{(x+y)} \mid x, y \in S\} \cup \{\frac{\{x,y\}}{(x \times y)} \mid x, y \in S\} \ . \quad \blacksquare$$

We see here an important distinction between inductive definitions in set theory and inductive definitions in Type Theory (see Section 1.2.1). In Type Theory, we think of inductive definitions as a method for generating new types. In contrast, in set theory all sets already exist. To form the rule set which defines the natural numbers in Examples 1.2(i), we already need the natural numbers! In this sense, rule sets are more like predicates defined as inductive families in Type Theory. Kleene [1952] makes the distinction between *fundamental* and *non-fundamental* inductive definitions, where rule sets give rise to the latter. For our purposes, this is not a problem, since we are interested in set theory (and inductive definitions therein) mostly as a model for Type Theory, and not as a foundational theory in itself.

There is an alternative presentation of rule sets on A as monotone operators on $\mathcal{P}(A)$, i.e. functions $\varphi : \mathcal{P}(A) \to \mathcal{P}(A)$ such that if $X \subseteq Y$ then $\varphi(X) \subseteq \varphi(Y)$. Given a monotone operator $\varphi : \mathcal{P}(A) \to \mathcal{P}(A)$, there is a corresponding rule set

$$\Phi_\varphi = \{\frac{X}{y} \mid X \subseteq A, y \in \varphi(X)\} \ ,$$

and we then have that $Y \subseteq A$ is Φ_φ-closed if and only if $\varphi(Y) \subseteq Y$. Conversely, given a rule set Φ on A, we define a monotone operator $\varphi_\Phi : \mathcal{P}(A) \to \mathcal{P}(A)$ by

$$\varphi_\Phi(Y) = \{x \in A \mid \frac{X}{x} \in \Phi \text{ for some } X \subseteq Y\} \ .$$

Then $Y \subseteq A$ is Φ-closed if and only if $\varphi_\Phi(Y) \subseteq Y$. Hence the set inductively defined by Φ can equivalently be described as the least set Y such that $\varphi_\Phi(Y) \subseteq Y$. This suggests the following transfinite construction of $\mathcal{I}(\Phi)$ "from below":

Proposition 1.3 Let Φ be a rule set on A. Define by transfinite recursion

$$\begin{aligned} \varphi^0 &= \emptyset \\ \varphi^{\alpha+1} &= \varphi_\Phi(\varphi^\alpha) \\ \varphi^\lambda &= \bigcup_{\beta<\lambda} \varphi_\Phi(\varphi^\beta) \qquad \lambda \text{ limit} \end{aligned}$$

Then there is some $\kappa \leq |A|$ such that $\mathcal{I}(\Phi) = \varphi^\kappa$, and $\mathcal{I}(\Phi)$ is the least fixed point of φ_Φ.

Proof. Let κ be the least ordinal such that $\varphi^{\kappa+1} = \varphi^\kappa$. Such a $\kappa \leq |A|$ must exist, since φ_Φ is monotone; hence, if we have not reached a fixed point yet, we are adding at least one new element at each iteration. Since $\varphi^\alpha \subseteq A$ for all α, we can do this at most $|A|$ times. We then have $\varphi_\Phi(\varphi^\kappa) = \varphi^\kappa$, in particular $\varphi_\Phi(\varphi^\kappa) \subseteq \varphi^\kappa$ and hence $\mathcal{I}(\Phi) \subseteq \varphi^\kappa$ since $\mathcal{I}(\Phi)$ is the least Φ-closed set. In the other direction, we can easily prove $\varphi^\alpha \subseteq \mathcal{I}(\Phi)$ for all α by transfinite induction, since φ_Φ is monotone. Hence $\mathcal{I}(\Phi) = \varphi^\kappa$, and φ^κ is the least fixed point of φ_Φ by construction. □

From a categorical perspective, φ^α is the initial sequence of the functor $\varphi_\Phi : \mathcal{P}(A) \to \mathcal{P}(A)$, and the proposition says that the sequence stabilises. The bound $\kappa \leq |A|$ is fine if we already know that we are dealing with a rule set on A. However, we often do not know a base set beforehand, but only have some large set V with the appropriate closure properties which we hope will contain all the types of the model. A bound $\leq |V|$ is then not good enough, as iterating $|V|$ times might make us end up with a set which is too large to be contained in V. Thankfully, we can often give more precise bounds.

Definition 1.4 Let κ be a cardinal. An operator φ is *κ-based* if $x \in \varphi(X)$ implies $x \in \varphi(Y)$ for some $Y \subseteq X$ of cardinality $< \kappa$. ■

Examples 1.5

(i) If $\varphi(X) = A \to X$ for some set A, then φ is κ-based for all $\kappa > |A|$.

(ii) Monotone operators φ corresponding to a finitary rule sets, i.e. rule sets where each set of premises is finite, are ω-based. ■

Proposition 1.6 Let φ_Φ be a κ-based monotone operator for a regular cardinal κ. Then $\mathcal{I}(\Phi) = \varphi^\kappa$.

Proof. We need to show $\varphi_\Phi(\varphi^\kappa) = \varphi^\kappa$. The direction $\varphi_\Phi(\varphi^\kappa) \supseteq \varphi^\kappa$ is clear by monotonicity of φ_Φ. For the other direction, let $x \in \varphi_\Phi(\varphi^\kappa)$. Then $x \in \varphi_\Phi(Y)$ for some $Y \subseteq \varphi^\kappa$ of cardinality $< \kappa$. But by the regularity of κ, then already $Y \subseteq \varphi^\beta$ for some $\beta < \kappa$ and $x \in \varphi_\Phi(Y) \subseteq \varphi_\Phi(\varphi^\beta) = \varphi^{\beta+1} \subseteq \varphi^\kappa$. □

1.3 Overview

The rest of the thesis is structured in the following way:

- **Chapter 2** introduces Martin-Löf type theory, including the notation and conventions we will use for the rest of the thesis. The content is entirely standard.
- In **Chapter 3**, we give several examples of inductive-inductive definitions. We then present a general finite axiomatisation of such definitions which extends the type theory introduced in Chapter 2. There are no deep theorems in this chapter, instead we introduce the object of study for the chapters to come.
- **Chapter 4** gives an alternative, categorical characterisation of the elimination rules for inductive-inductive definitions. Along the way, we develop a theory of generic eliminators, using the concept of a Category with Families. The main result is an equivalence between the elimination rules and the existence of an initial object in a certain category (Theorem 4.43).
- In **Chapter 5**, the semantics of inductive-inductive definitions is considered. We give two different models: a set-theoretic one (Theorem 5.14), and an interpretation of inductive-inductive definitions as indexed inductive definitions (Theorem 5.39). The latter translation is simplified by first giving an "inductive-inductive container" semantics (Corollaries 5.19 and 5.25) for inductive-inductive definitions.
- The theory as presented in Chapter 3 is not strong enough to cover all the examples of inductive-inductive definitions that have appeared in the literature. In **Chapter 6**, we consider several extensions that makes it possible to handle the other examples, and prove that the set-theoretic model from Chapter 5 can be extended to handle the extended theory (Theorem 6.15).
- **Chapter 7** puts the theory to use and considers two larger examples of uses of inductive-inductive definitions: Conway's surreal numbers and another variant of inductive-recursive definitions which is attractive from a categorical point of view.
- Finally, **Chapter 8** concludes and outlines plans for future research.

Publications

Parts of this thesis have been published in peer-reviewed conferences:

(i) **Inductive-inductive definitions**. With Anton Setzer. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 2010. [Nordvall Forsberg and Setzer, 2010].

(ii) **A categorical semantics for inductive-inductive definitions**. With Thorsten Altenkirch, Peter Morris, and Anton Setzer. In Andrea Corradini, Bartek Klin, and Corina Cirstea, editors, *Conference on Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 70 – 84. Springer, 2011. [Altenkirch et al., 2011].

(iii) **A finite axiomatisation of inductive-inductive definitions**. With Anton Setzer. In Ulrich Berger, Hannes Diener, Peter Schuster, and Monika Seisenberger, editors, *Logic, Construction, Computation*, volume 3 of *Ontos mathematical logic*, pages 259 – 287. Ontos Verlag, 2012. [Nordvall Forsberg and Setzer, 2012].

(iv) **Positive inductive-recursive definitions**. With Neil Ghani and Lorenzo Malatesta. In Reiko Heckel, and Stefan Milius, editors, invited paper *Conference on Algebra and Coalgebra in Computer Science*, volume 8089 of *Lecture Notes in Computer Science*, pages 19 – 33. Springer, 2013. [Ghani et al., 2013a].

My contributions to the publications are:

(i) I wrote the entire article, and did most of ·he work for it (in collaboration with Anton Setzer). Most of the material has been superceded by Article (iii), but traces can still be found in Chapters 3 and 5.

(ii) I wrote the entire article. The initial idea came from Thorsten Altenkirch and Peter Morris, and was then developed by me. Parts of Chapter 4 is based on this article.

(iii) I wrote the entire article. This is a refinement of Article (i), taking ideas from Article (ii) into account. Chapter 3 is based on this article.

(iv) The article is jointly written by Lorenzo Malatesta and me (except for Section 4, which is written by Neil Ghani), and is truly joint work, after an initial idea from Lorenzo Malatesta and Neil Ghani. The material developed in the article makes up the first half of Chapter 7.

Chapter 2

Martin-Löf Type Theory

Contents

This chapter contains the necessary background material for the rest of the thesis. We introduce Martin-Löf Type Theory, the system we will be working in and extending. As a large part of the thesis applies categorical methods, we also discuss the status of category theory in and for Type Theory.

2.1 An intuitionistic theory of types

We present the monomorphic version of Martin-Löf's Type Theory, presented using the Logical Framework. This corresponds to the presentation in Nordström et al. [1990, Part III]. As is usually done, we formulate the rules in the style of natural deduction [Prawitz, 1965]. To make the presentation easier to digest, we split it up into several definitions. The rules of Type Theory are all the rules presented in Axioms 2.1 to 2.5 and 2.7 in this section.

Martin-Löf Type Theory is a theory about types and their elements. We can think of types and terms in many ways: as sets and elements, spaces and points, propositions and proofs, or specifications and programs, for instance. As a formal theory, Type Theory contains rules for making judgements of the following forms:

- A is a type, written $\boxed{A \text{ type}}$.
- A and B are equal types, written $\boxed{A = B}$.
- a is a term of the type A, written $\boxed{a : A}$.

- a and b are equal terms of the type A, written $\boxed{a = b : A}$.

We will later introduce a type Set which we think of as containing small sets, which will lead to further (derived) judgements such as $A : \mathsf{Set}$ and $a : A$ for $A : \mathsf{Set}$.

In general, a judgement is made under assumptions, which are collected in a *context* Γ. Hence we will also need a fifth judgement, namely that a given context Γ is well-formed, i.e. consists of distinct variables of well-formed types, written $\boxed{\Gamma\ \mathsf{context}}$. We write the context in front of the other judgements, and separate the context and the judgement with a turnstile, like so:

$$\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash A = B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a = b : A$$

Following Troelstra [1987], when the context Γ is unchanged from the hypothesis to the conclusion in a rule, we will omit both Γ and the turnstile $\vdash$. We will take similar notational shortcuts in the running text, so that for instance a sentence "let $x : A \ldots$" should be understood in an arbitrary context Γ. A context consists of a telescope [de Bruijn, 1991] of typing judgements

$$x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, x_2, \ldots, x_{n-1})$$

i.e. we require that each A_i is a type in the smaller context $x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$:

$$\begin{gathered} \vdash A_1\ \mathsf{type} \\ x_1 : A_1 \vdash A_2(x_1)\ \mathsf{type} \\ \vdots \\ x_1 : A_1, x_2 : A_2, \ldots, x_{n-1} : A_{n-1} \vdash A_n(x_1, x_2, \ldots, x_{n-1})\ \mathsf{type}\ . \end{gathered}$$

As a notational aid, we have indicated the free variables of the types in the context in brackets. Formally, we can inductively define context validity by the following two rules, where we write $\diamond$ for the empty context (we will omit $\diamond$ in judgements $\diamond \vdash \mathcal{J}$ and simply write $\vdash \mathcal{J}$):

Axiom 2.1 (Valid contexts) Valid contexts are inductively generated by the following two rules:

$$\frac{}{\diamond\ \mathsf{context}} \qquad \frac{\Gamma\ \mathsf{context} \qquad \Gamma \vdash A\ \mathsf{type}}{(\Gamma, x : A)\ \mathsf{context}}\ (x \notin FV(\Gamma))$$

Here, the side condition $x \notin FV(\Gamma)$ means that x is not among the free variables declared in Γ. In a fully formalised account, one could use for instance de Bruijn indices [de Bruijn, 1972] to get rid of this side condition. ■

Note how this definition refers to the definition of A type, which in turn refers to the definition of contexts. As we will see in Section 3.1, this means that the very definition of Type Theory has a flavour of the kind of definitions that this thesis is studying.

Given a type B depending on $x : A$ (i.e. $x : A \vdash B(x)$ type) and a term $a : A$, we write $B[x \mapsto a]$ for B where we have substituted every free occurrence of x by a, and similarly for a term $x : A \vdash b(x) : B(x)$. We can extend this to the simultaneous substitution of n terms

$$\begin{gathered}\Gamma \vdash a_1 : A_1 \\ \Gamma \vdash a_2 : A_2[x_1 \mapsto a_1] \\ \vdots \\ \Gamma \vdash a_n : A_n[x_1 \mapsto a_1, \ldots, x_{n-1} \mapsto a_{n-1}]\end{gathered}$$

into a type $x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1}) \vdash B(x_1, \ldots, x_n)$ type. After we have introduced function types in Section 2.1.2, we can relax the notation a bit and simply write $\Gamma \vdash B(a_1, \ldots, a_n)$ type for $\Gamma \vdash B[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ type. We identify types and terms up to α-conversion, i.e. up to renaming of variables. In general, we employ the Barendregt convention [Barendregt, 1984] and make sure that we always choose free variable names distinct from bound ones.

We now present the rest of the rules. The rules can be divided into four main groups: general rules for equality and substitution, rules for the function type, rules for set formation and rules for some basic set formers. The other set formers will be introduced via the principle of inductive-inductive definitions in Chapter 3. The rules for the different type and set formers follow a common pattern. They can be further categorised to be of one of the following forms:

- The *formation rule* for A describes when we may infer that A is a type or a set.
- The *introduction rules* for A describe how to introduce canonical elements of type A. This corresponds to listing the constructors for A.
- The *elimination rules* for A describe how to prove a proposition about an arbitrary element of type A. This corresponds to primitive recursion or proof by induction. The "target type" $P : A \to \mathrm{Set}$ of the elimination rule is called the *motive* of the rule.
- The *computation rules* describe the computational behaviour of the eliminators.

2.1.1 General equality and substitution rules

The following rules form the equality, substitution and variable assumption rules of Type Theory. They are entirely standard.

Axioms 2.2 Equality of types is an equivalence relation:

$$\frac{A \text{ type}}{A = A} \qquad \frac{A = B}{B = A} \qquad \frac{A = B \quad B = C}{A = C}$$

Equality of elements is an equivalence relation:

$$\frac{a : A}{a = a : A} \qquad \frac{a = b : A}{b = a : A} \qquad \frac{a = b : A \quad b = c : A}{a = c : A}$$

Typing and equality is well-behaved:

$$\frac{a : A \qquad A = B}{a : B} \qquad\qquad \frac{a = b : A \qquad A = B}{a = b : B}$$

Substitution interacts well with equality:

$$\frac{\Gamma, x : A \vdash B(x)\ \text{type} \qquad \Gamma \vdash a : A}{\Gamma \vdash B[x \mapsto a]\ \text{type}} \qquad \frac{\Gamma, x : A \vdash B(x) = D(x) \qquad \Gamma \vdash a = c : A}{\Gamma \vdash B[x \mapsto a] = D[x \mapsto c]}$$

$$\frac{\Gamma, x : A \vdash b(x) : B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash b[x \mapsto a] : B(a)} \qquad \frac{\Gamma, x : A \vdash b(x) = d(x) : B(x) \qquad \Gamma \vdash a = c : A}{\Gamma \vdash b[x \mapsto a] = d[x \mapsto c] : B(a)}$$

Assumption:

$$\frac{\Gamma, x : A, \Delta\ \text{context}}{\Gamma, x : A, \Delta \vdash x : A}$$ ■

Notice in the assumption rule that for $\Gamma, x : A, \Delta$ context to be valid, we need $\Gamma \vdash A$ type.

2.1.2 Set and function types

We now introduce a type (universe) Set of small types. Most of the time, we will work with small types only, and only use the large types of the logical framework to simplify the description of e.g. the elimination rules for sets. Notable exceptions are the universes SP_A and SP_B of codes for inductive-inductive definitions that we will introduce in Chapter 3. Since certain codes quantify over arbitrary small sets, and we want a predicative theory, we cannot make the universe of codes itself a small type.

Axioms 2.3 (The type of sets) Formation rule for Set:

$$\frac{\Gamma\ \text{context}}{\Gamma \vdash \mathsf{Set}\ \text{type}}$$

The elements of a set form a type:

$$\frac{A : \mathsf{Set}}{\mathsf{El}(A)\ \text{type}}$$

Congruence for El:

$$\frac{A = B : \mathsf{Set}}{\mathsf{El}(A) = \mathsf{El}(B)}$$ ■

Since no confusion is possible, we will allow ourselves to write $a : A$ as a shorthand for $a : \mathsf{El}(A)$ if $A : \mathsf{Set}$. In a way, we are treating the large universe à la Tarski (Set, El) as a universe à la Russel [Martin-Löf, 1984]. Morally, we are employing a particularly simple form of coercive subtyping [Luo et al., 2012] Set $\leq_{\mathsf{El}}$ type, were it not for the fact that type is not an object in our theory (see also Luo [2012]).

Next we introduce the rules for function types and function sets, also called Π-types. We deliberately use the same notation for both, so that we can get away with stating most rules only once. Of course, officially there are two sets of rules, one for function types and one for function sets. Function types are essential for describing families of sets in the theory.

Axioms 2.4 (Function types and dependent functions) Function type formation:

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

Function type introduction:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda(x{:}A).\, b(x) : (x : A) \to B(x)}$$

Function application:

$$\frac{f : (x : A) \to B(x) \qquad a : A}{f(a) : B[x \mapsto a]}$$

Computation (β equality):

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

Equalities with and under binders (η and ξ equality):

$$\frac{f : (x : A) \to B(x)}{\lambda(x{:}A).\, f(x) = f : (x : A) \to B(x)} \qquad \frac{\Gamma, x : A \vdash f(x) = g(x) : B(x)}{\Gamma \vdash \lambda(x{:}A).\, f(x) = \lambda(x{:}A).\, g(x) : (x : A) \to B(x)}$$

Function type congruences:

$$\frac{\Gamma \vdash A = C \qquad \Gamma, x : A \vdash B(x) = D(x)}{\Gamma \vdash ((x : A) \to B(x)) = ((x : C) \to D(x))} \qquad \frac{f = g : (x : A) \to B(x) \qquad a = b : A}{f(a) = g(b) : B[x \mapsto a]}$$ ∎

We write $A \to B$ for $(x : A) \to B$ if x does not occur in B. Furthermore, we will sometimes write $\lambda x.\, e$ for $\lambda(x{:}A).\, e$ if the type A of x can be inferred from context, and we write repeated application as $f(x_1, \dots, x_k)$. Now that we have introduced function types and the type Set, we can introduce sets by defining different constants, and asserting equalities between elements in the sets. Note first that with the rules for function introduction and application, there is no essential difference between a family of sets $Y : (x : X) \to \text{Set}$ and a set $Y(x)$ with a free variable $x : X$ anymore: Given $Y : (x : X) \to \text{Set}$, we can derive

$$\frac{\Gamma, x : X \vdash Y : (x : X) \to \text{Set} \qquad \overline{\Gamma, x : X \vdash x : X}}{\Gamma, x : X \vdash Y(x) : \text{Set}}$$

using application, and given $\Gamma, x : X \vdash Y(x) : \mathsf{Set}$, we derive

$$\cfrac{\cfrac{\Gamma, x : X \vdash Y(x) : \mathsf{Set}}{\Gamma \vdash \lambda(x{:}X).\,Y(x) = Y : (x : X) \to \mathsf{Set}}}{\Gamma \vdash Y : (x : X) \to \mathsf{Set}}$$

using introduction and the η-rule.

Thus, instead of introducing the function sets as we just did, we could have introduced constants

$$\begin{aligned}
\Pi &: (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to \mathsf{Set} \\
\lambda &: (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to ((x : X) \to Y(x)) \to \Pi(X, Y) \\
\mathsf{apply} &: (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to \Pi(X, Y) \to (x : X) \to Y(x)
\end{aligned}$$

and asserted the equality

$$\mathsf{apply}(X, Y, \lambda(X, Y, b), a) = b(a) : B(a)$$

The formation, introduction, application, computation and congruence rules above would then be derivable. We would still have to assert the η and ξ rules. Even if we continue to present the rest of the rules in natural deduction style, we officially consider this to be a shorthand for the constant approach above. This way, there is no further need to state any more congruence rules, as they are included in the congruence rules for function types.

2.1.3 Sets as type constants

We now introduce some further sets by postulating the existence of certain constants. We can get away with quite a small collection of sets: the empty set, the unit set, the Booleans and Σ-types. In particular, at this stage we do not need to introduce any infinite or "properly inductive sets", since we will get them via inductive-inductive definitions in Chapter 3.

Axioms 2.5 (The sets **0**, **1** and **2**) Formation rules:

$$\frac{\Gamma\ \mathsf{context}}{\Gamma \vdash \mathbf{0} : \mathsf{Set}} \qquad \frac{\Gamma\ \mathsf{context}}{\Gamma \vdash \mathbf{1} : \mathsf{Set}} \qquad \frac{\Gamma\ \mathsf{context}}{\Gamma \vdash \mathbf{2} : \mathsf{Set}}$$

Introduction rules:

$$\frac{}{\star : \mathbf{1}} \qquad \frac{}{\mathsf{tt} : \mathbf{2}} \qquad \frac{}{\mathsf{ff} : \mathbf{2}}$$

The elimination rule for **0** and the η rule for **1**:

$$\frac{P : \mathbf{0} \to \mathsf{Set} \qquad x : \mathbf{0}}{!_P(x) : P(x)} \qquad \frac{x : \mathbf{1}}{x = \star : \mathbf{1}}$$

Elimination rule for **2**:

$$\frac{\Gamma, x : \mathbf{2} \vdash P(x)\ \mathsf{type} \qquad \Gamma \vdash a : P(\mathsf{tt}) \qquad \Gamma \vdash b : P(\mathsf{ff}) \qquad y : \mathbf{2}}{\Gamma \vdash \mathsf{if}_P\ y\ \mathsf{then}\ a\ \mathsf{else}\ b : P(y)}$$

Computation rules for **2**:

$$\begin{aligned}\mathsf{if}_P\ \mathsf{tt}\ \mathsf{then}\ a\ \mathsf{else}\ b &= a : P(\mathsf{tt})\\ \mathsf{if}_P\ \mathsf{ff}\ \mathsf{then}\ a\ \mathsf{else}\ b &= b : P(\mathsf{ff})\end{aligned}$$ ■

We will write ! and if · then · else · for $!_P$ and if_P · then · else · respectively if P can be inferred from the context.

Proposition 2.6 We can define a constant $\mathsf{elim}_\mathbf{1}$ such that the elimination rule

$$\frac{P : \mathbf{1} \to \mathsf{Set} \qquad m : P(\star) \qquad x : \mathbf{1}}{\mathsf{elim}_\mathbf{1}(P, m, x) : P(x)}$$

with computation rule

$$\mathsf{elim}_\mathbf{1}(P, m, \star) = m : P(\star)$$

is derivable.

Proof. Since $x = \star$ by the η rule, the computation rule determines $\mathsf{elim}_\mathbf{1}$ uniquely and can be taken as a definition. □

We would now like to draw attention to the only perhaps unusual feature of the type theory presented here: we have large elimination for Booleans. This will make it possible to simplify the axiomatisation in Chapter 3 slightly (see Section 3.2.3.5). For the same reason, we added the η rule for the unit type. We stress that these features are not required for the development to come; we could work in a type theory without η equality and large elimination for Booleans, if we pay the higher price of adding the rules we now can derive to our axiomatisation.

We now introduce the last set former we need, the set of dependent pairs, also called Σ-types:

Axioms 2.7 (Dependent pairs) Formation rule:

$$\frac{\Gamma \vdash A : \mathsf{Set} \qquad \Gamma, x : A \vdash B(x) : \mathsf{Set}}{\Gamma \vdash (\Sigma x : A)B(x) : \mathsf{Set}}$$

Introduction rule:

$$\frac{a : A \qquad b : B(a)}{\langle a, b\rangle : (\Sigma x : A)B(x)}$$

Elimination rules (projections):

$$\frac{p : (\Sigma x : A)B(x)}{\mathsf{fst}(p) : A} \qquad \frac{p : (\Sigma x : A)B(x)}{\mathsf{snd}(p) : B(\mathsf{fst}(p))}$$

Surjective pairing (η equality):

$$\frac{p : (\Sigma x : A)B(x)}{p = \langle \mathsf{fst}(p), \mathsf{snd}(p)\rangle : (\Sigma x : A)B(x)}$$

Computation rules:

$$\begin{aligned}\mathsf{fst}(\langle a,b\rangle) &= a : A\\ \mathsf{snd}(\langle a,b\rangle) &= b : B(a)\end{aligned} \quad ■$$

If $B(x)$ does not depend on x, we write $A \times B$ for $(\Sigma x : A)B(x)$. We have presented both Π-types and Σ-types in a *negative* way (see e.g. Zeilberger [2009]), i.e. characterised by their observations – applications and projections. However, since we also included η rules, we can actually recover the positive point of view as well:

Proposition 2.8 We can define a constant split such that the elimination rule

$$\frac{P : (\Sigma x : A)B(x) \to \mathsf{Set} \quad m : (a : A) \to (b : B(a)) \to P(\langle a,b\rangle) \quad y : (\Sigma x : A)B(x)}{\mathsf{split}(P,m,y) : P(y)}$$

with computation rule

$$\mathsf{split}(P,m,\langle a,b\rangle) = m(a,b) : P(\langle a,b\rangle)$$

is derivable.

Proof. We define $\mathsf{split}(P,m,y) := m(\mathsf{fst}(y),\mathsf{snd}(y)) : P(\langle\mathsf{fst}(y),\mathsf{snd}(y)\rangle)$. This is type-correct since $\langle\mathsf{fst}(y),\mathsf{snd}(y)\rangle = y$ by surjective pairing, and the computation rule holds by the computation rules for fst and snd. □

We can of course also define fst and snd in terms of split, had we chosen to introduce split as the basic notion. We can also prove the η rule up to propositional equality (see Section 2.1.6). In fact, such a propositional η rule and projections are equivalent to the general elimination rule. Garner [2009] shows the same result for Π-types, i.e. propositional η and application is equivalent to elimination. Garner also shows that application without η does not entail the general elimination rule.

2.1.4 Derived rules and meta-theoretical properties

One set former that seems to be missing from the previous section is the sum, or disjoint union, of two sets $A + B$. This is not a problem, since large elimination for Booleans, which we need for other reasons, allows us to also construct sums in the following way:

Proposition 2.9 We can define the disjoint union of two sets A and B satisfying the following rules.

Formation rule:

$$\frac{A : \mathsf{Set} \quad B : \mathsf{Set}}{A + B : \mathsf{Set}}$$

Introduction rules:

$$\frac{a : A}{\mathsf{inl}(a) : A + B} \qquad \frac{b : B}{\mathsf{inr}(b) : A + B}$$

Elimination rule:

$$\frac{P : (x : A + B) \to \mathsf{Set} \quad \begin{array}{c} f : (x : A) \to P(\mathsf{inl}(x)) \\ g : (y : B) \to P(\mathsf{inr}(y)) \end{array} \quad c : A + B}{[f, g]_P(c) : P(c)}$$

Computation rules:

$$\begin{aligned} [f, g]_P(\mathsf{inl}(a)) &= f(a) : P(\mathsf{inl}(a)) \\ [f, g]_P(\mathsf{inr}(b)) &= g(b) : P(\mathsf{inr}(b)) \end{aligned} \quad .$$

Proof. We can define $A + B := (\Sigma x : \mathbf{2})(\mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B)$ and then

$$\begin{aligned} \mathsf{inl}(a) &:= \langle \mathsf{tt}, a \rangle \\ \mathsf{inr}(b) &:= \langle \mathsf{ff}, b \rangle \\ [f, g]_P(c) &:= \mathsf{split}(P, \lambda x.\, \mathsf{if}_{\lambda z.\, (y : (\mathsf{if}\ z\ \mathsf{then}\ A\ \mathsf{else}\ B)) \to P(\langle z, y \rangle)}\ x\ \mathsf{then}\ f\ \mathsf{else}\ g, c) \end{aligned}$$

We easily check that e.g.

$$\begin{aligned} [f, g]_P(\mathsf{inl}(a)) &= \mathsf{split}(P, \lambda x.\, \mathsf{if}\ x\ \mathsf{then}\ f\ \mathsf{else}\ g, \langle \mathsf{tt}, a \rangle) \\ &= (\mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ f\ \mathsf{else}\ g)(a) \\ &= f(a) \end{aligned}$$

and similarly for $[f, g]_P(\mathsf{inr}(b)) = g(b)$. □

In the presence of extensional identity types (see Section 2.1.6), large elimination is not needed. In this setting, Troelstra [1983] encodes sums using Σ-types, function types, identity types and Booleans (without large elimination). Note also that Booleans can be encoded using sums: $\mathbf{2} = \mathbf{1} + \mathbf{1}$, but without large elimination.

We now give the main meta-theoretical results about the Type Theory presented. The proofs are standard, and can be found in e.g. Goguen [1994]; Luo [1994]; Werner [1994]. Some of them serve as sanity checks for the Type Theory, others will be implicitly used in the rest of this thesis. It is not such a bold conjecture that these properties will continue to hold when we extend the theory in the coming chapters.

We write $\Gamma \vdash \mathcal{J}$ for an arbitrary judgement of the form Γ context, $\Gamma \vdash A$ type or $\Gamma \vdash t : A$.

Proposition 2.10 (Weakening) Let Γ, Δ be a valid context such that $\Gamma \vdash A$ type, and assume $x \notin FV(\Gamma)$. If $\Gamma, \Delta \vdash \mathcal{J}$ is derivable then so is $\Gamma, x : A, \Delta \vdash \mathcal{J}$. □

Proposition 2.11 (Substitution) Let $\Gamma, x : A, \Delta$ be a valid context and $\Gamma \vdash t : A$. If the judgement $\Gamma, x : A, \Delta \vdash \mathcal{J}$ is derivable, then so is $\Gamma, \Delta[x \mapsto t] \vdash \mathcal{J}[x \mapsto t]$. □

Proposition 2.12 (Sanity checks)

(i) If $\Gamma \vdash \mathcal{J}$, then Γ context.

(ii) If $\Gamma \vdash a : A$ or $\Gamma \vdash A = B$, then $\Gamma \vdash A$ type.

(iii) If $\Gamma \vdash a = b : A$, then $\Gamma \vdash a : A$. □

Proposition 2.13 (Unicity of typing) If $\Gamma \vdash a : A$ and $\Gamma \vdash a : B$ then $\Gamma \vdash A = B$. □

Theorem 2.14 (Strong normalisation) By directing the computation rules from left to right, one obtains a strongly normalising rewrite system. □

Theorem 2.15 (Decidable type checking) Given a judgement $\Gamma \vdash \mathcal{J}$, it is decidable if there is a derivation of $\Gamma \vdash \mathcal{J}$ or not. □

2.1.5 The Curry-Howard isomorphism: propositions-as-types

Martin-Löf Type Theory is "intended to be a full scale system for formalising intuitionistic mathematics" [Martin-Löf, 1972], but so far, we have introduced something more along the lines of a programming language. Logic and reasoning is reintroduced via the Curry-Howard isomorphism [Curry, 1934; Curry and Feys, 1958; Howard, 1969] (see also Scott [1970]). Propositions are identified with types consisting of their proofs, as can be seen in Table 2.1.

Table 2.1: Propositions as types.

Proposition	Type
$\bot$	$\mathbf{0}$
$\top$	e.g. $\mathbf{1}$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \Rightarrow B$	$A \to B$
$(\exists x : A)B(x)$	$(\Sigma x : A)B(x)$
$(\forall x : A)B(x)$	$(x : A) \to B(x)$

This is in accordance with the Brouwer-Heyting-Kolmogorov interpretation (see e.g. Troelstra and van Dalen [1988]) of intuitionistic logic:

- There is no proof of $\bot$, since there is no element of type $\mathbf{0}$.
- A proof $p = \langle q, r \rangle$ of $A \wedge B$ consists of a proof q of A and a proof r of B.
- A proof p of $A \vee B$ is either of the form $\mathsf{inl}(q)$ where q is a proof of A, or of the form $\mathsf{inr}(r)$ where r is a proof of B. Hence a proof of $A \vee B$ is either a proof of A or a proof of B, and we can tell which one it is.
- A proof of $A \Rightarrow B$ is a function that transforms proofs of A into proofs of B.
- A proof $p = \langle a, q \rangle$ of $(\exists x : A)B(x)$ is a witness $a : A$, together with a proof q of $B(a)$.

- A proof p of $(\forall x : A)B(x)$ is a function which given $a : A$ produces a proof of $B(a)$.

The only thing lacking are atomic propositions, except for $\top$ and $\bot$. We will introduce equality in the next section, and a lot more data types corresponding to atomic propositions in Chapter 3.

Remark 2.16 The correspondence between propositions and types, and proofs and programs is called an isomorphism, and not just a bijection, since it also preserves reductions: the computation rules we have specified correspond exactly to proof normalisation from proof theory [Girard et al., 1989].

Example 2.17 For types A : Set and B : Set, and a predicate $P : A \to B \to \mathsf{Set}$, let us prove the proposition

$$(\exists x : A)(\forall y : B)P(x,y) \Rightarrow (\forall y : B)(\exists x : A)P(x,y)$$

in Type Theory. By the propositions-as-types principle, this corresponds to constructing a term of type

$$(\Sigma x : A)((y : B) \to P(x,y)) \to (y : B) \to (\Sigma x : A)P(x,y)$$

Here is the typing derivation for such a term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash p : (\Sigma x : A)((y : B) \to P(x,y))}}{\Gamma \vdash \mathsf{fst}(p) : A} \qquad \dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash p : (\Sigma x : A)((y : B) \to P(x,y))}}{\mathsf{snd}(p) : (y : B) \to P(\mathsf{fst}(p),y)} \qquad \dfrac{\mathcal{D}}{\Gamma \vdash b : B}}{\Gamma \vdash \mathsf{snd}(p)(b) : P(\mathsf{fst}(p),b)}}{\Gamma \vdash \langle \mathsf{fst}(p), \mathsf{snd}(p)(b)\rangle : (\Sigma x : A)P(x,b)}}{p : (\Sigma x : A)((y : B) \to P(x,y)) \vdash \lambda b.\, \langle \mathsf{fst}(p), \mathsf{snd}(p)(b)\rangle : (y : B) \to (\Sigma x : A)P(x,y)}}{\vdash \lambda p.\, \lambda b.\, \langle \mathsf{fst}(p), \mathsf{snd}(p)(b)\rangle : (\Sigma x : A)((y : B) \to P(x,y)) \to (y : B) \to (\Sigma x : A)P(x,y)}$$

where we have written $\Gamma := [p : (\Sigma x : A)((y : B) \to P(x,y)), b : B]$ and $\mathcal{D}$ is the following derivation of Γ context:

$$\dfrac{\dfrac{\dfrac{}{\diamond\ \mathsf{context}} \qquad \dfrac{\vdash A : \mathsf{Set} \qquad \dfrac{x : A \vdash B : \mathsf{Set} \qquad x : A, y : B \vdash P(x,y) : \mathsf{Set}}{x : A \vdash (y : B) \to P(x,y) : \mathsf{Set}}}{\vdash (\Sigma x : A)((y : B) \to P(x,y)) : \mathsf{Set}}}{p : (\Sigma x : A)((y : B) \to P(x,y))\ \mathsf{context}} \qquad \ldots \vdash B : \mathsf{Set}}{\Gamma\ \mathsf{context}}$$

By Theorem 2.15, the whole derivation can be reconstructed from the proof term

$$\lambda p.\, \lambda b.\, \langle \mathsf{fst}(p), \mathsf{snd}(p)(b)\rangle : (\Sigma x : A)((y : B) \to P(x,y)) \to (y : B) \to (\Sigma x : A)P(x,y) \ ,$$

and so, we will not display the derivation trees in the rest of this thesis. ∎

2.1.6 Equality and identity types

We now introduce another type former, namely the identity type $x \equiv_A y$ whose inhabitants are proofs that x and y are equal elements of type A. Actually, this is not necessary: the identity type is an indexed inductive definition, and can hence be defined using the principle of inductive-inductive definitions in Chapter 3. When giving the axiomatisation of inductive-inductive definitions, we will be careful not to make use of the identity type, so that we do not need to add it to our theory. Nevertheless, we still need to develop some standard infrastructure for making use of identity types, and will in certain parts go beyond the inductively defined identity type and use extensional equality, and so, we introduce the type former now.

2.1.6.1 Intensional and extensional Type Theory

We introduce the (intensional) identity type with the following rules:

Axiom 2.18 Formation rule:

$$\frac{A\ \mathsf{type} \qquad x, y : A}{x \equiv_A y}$$

Introduction rule:

$$\frac{}{\mathsf{refl} : x \equiv_A x}$$

Elimination rule:

$$\frac{P : (x, y : A) \to x \equiv_A y \to \mathsf{Set} \qquad \mathsf{step_{refl}} : (x : A) \to P(x, x, \mathsf{refl}) \qquad \begin{array}{c} x, y : A \\ p : x \equiv_A y \end{array}}{\Gamma \vdash \mathsf{elim}_{\equiv}(P, \mathsf{step_{refl}}, x, y, p) : P(x, y, p)}$$

Computation rule:

$$\mathsf{elim}_{\equiv}(P, \mathsf{step_{refl}}, x, x, \mathsf{refl}) = \mathsf{step_{refl}}(x) : P(x, x, \mathsf{refl})$$ ∎

We see that by applying structural rules, we can make the rule

$$\frac{x = y : A}{\mathsf{refl} : x \equiv_A y}$$

admissible. The converse of this rule

$$\frac{\mathsf{refl} : x \equiv_A y}{x = y : A}\ \text{(EQUALITY REFLECTION)}$$

is called *equality reflection* and is not derivable in general [Hofmann and Streicher, 1998]. If equality reflection is added to the theory, then whenever $p : x \equiv_A y$ also $\mathsf{refl} : x \equiv_A y$. Hence we might as well add a coherence rule

$$\frac{p : x \equiv_A y}{p = \mathsf{refl} : x \equiv_A y}\ \text{(EQUALITY COHERENCE)}$$

The theory we get if we add these two rules is called extensional Type Theory. It is consistent (in fact, most known models for Type Theory validates equality reflection), but has some less than ideal properties: type checking is undecidable (intuitively because the the type checker might need to make up arbitrarily complex equality proofs to see that a term is well-typed) and the theory is not strongly normalising (intuitively because in an inconsistent context, we can e.g. fool the system to believe that $A = A \to A$ for some non-trivial set A, and then embed the untyped lambda calculus). We will use extensional Type Theory in some parts of this thesis, but will always point out when we do so.

There is a related concept of *function extensionality*, which says that extensionally equal functions are equal. In other words, function extensionality is the statement that there is a term $\mathsf{ext}_{f,g}$ for $f, g : (x : A) \to B(x)$ of type

$$\mathsf{ext}_{f,g} : ((x : A) \to f(x) \equiv_{B(x)} g(x)) \to f \equiv_{(x:A)\to B(x)} g$$

Function extensionality follows from extensional Type Theory (hence the name!), but is also available in more well-behaved type theories such as Observational Type Theory [Altenkirch et al., 2007] and Homotopy Type Theory[1] [The Univalent Foundations Program, 2013]. For this reason, we consider function extensionality acceptable, but will (not always successfully) try to avoid using it.

2.1.6.2 Properties of the intensional identity type

The identity type has all the structure one might expect:

Lemma 2.19 Propositional equality is symmetric, transitive and substitutive, i.e. there are terms

(i) $\mathsf{sym} : x \equiv_A y \to y \equiv_A x$ with $\mathsf{sym}(\mathsf{refl}) = \mathsf{refl}$,

(ii) $\mathsf{trans} : x \equiv_A y \to y \equiv_A z \to x \equiv_A z$ with $\mathsf{trans}(\mathsf{refl}, p) = p$, and

(iii) $\mathsf{subst} : (P : A \to \mathsf{Set}) \to x \equiv_A y \to P(x) \to P(y)$ with $\mathsf{subst}(P, \mathsf{refl}, x) = x$.

Proof. See e.g. Nordström et al. [1990, Chapter 8]. □

The next property of propositional equality is also standard, but we include its proof since it is similar to the proof of the next lemma, which is more technical.

Lemma 2.20 Propositional equality is a congruence with respect to every function, i.e. there is a term

$$\mathsf{cong}_{A,B} : (f : A \to B) \to (x : A) \to (y : A) \to x \equiv_A y \to f(x) \equiv_B f(y)$$

which further satisfies $\mathsf{cong}(f, x, x, \mathsf{refl}) = \mathsf{refl} : f(x) \equiv_B f(x)$.

[1] At the time of writing, it is an open question if Homotopy Type Theory is a well-behaved theory or not.

Proof. Given f, x, y and $p : x \equiv_A y$, apply the substitution principle for the identity type with motive $P(x') :\equiv f(x) \equiv_B f(x')$ to p. We need a term of type $P(x) = f(x) \equiv_B f(x)$, for which clearly refl suffices. Hence we can define

$$\mathsf{cong}_{A,B}(f,x,y,p) :\equiv \mathsf{subst}(\lambda(x'\!:\!A).\,f(x) \equiv_B f(x'), p, \mathsf{refl})$$

and we have

$$\mathsf{cong}_{A,B}(f,x,y,\mathsf{refl}) = \mathsf{subst}(\lambda(x'\!:\!A).\,f(x) \equiv_B f(x'), \mathsf{refl}, \mathsf{refl}) = \mathsf{refl}\ . \qquad \square$$

We will usually suppress the types A and B and the arguments x and y to $\mathsf{cong}_{A,B}$, as they can be inferred from the type of f and $p : x \equiv_A y$ respectively.

Lemma 2.21 For all A : Set, $B : A \to$ Set and C : Set, there is a term

$$\begin{aligned}\mathsf{cong}_2 : (f : (x : A) \to Bx \to C) \to (x : A) \to (y : A) \to (u : B(x)) \to (v : B(y)) \to \\ (p : x \equiv_A y) \to (q : \mathsf{subst}(B,p,u) \equiv_{B(y)} v) \to f(x,u) \equiv_C f(y,v)\end{aligned}$$

such that $\mathsf{cong}_2(f,x,x,u,u,\mathsf{refl},\mathsf{refl}) = \mathsf{refl} : f(x,u) \equiv_C f(x,u)$.

Proof. Given f, x, y, u, v, p and q, apply the elimination principle for the identity type with motive

$$\begin{aligned}&P(x',y',p') :\equiv \\ &\quad (u' : B(x')) \to (v' : B(y')) \to \mathsf{subst}(B,p',u') \equiv_{B(y')} v' \to f(x',u') \equiv_C f(y',v')\ .\end{aligned}$$

We need to give a term of type $P(x,x,\mathsf{refl})$, i.e. of type

$$(u' : B(x)) \to (v' : B(x)) \to u' \equiv_{B(x)} v' \to f(x,u') \equiv_C f(x,v')\ .$$

But that is exactly the type of $\mathsf{cong}_{B(x),C}(f(x))$ from Lemma 2.20. Furthermore,

$$\mathsf{cong}_2(f,x,x,u,u,\mathsf{refl},\mathsf{refl}) = \mathsf{cong}_{B(x),C}(f(x),u,u,\mathsf{refl}) = \mathsf{refl}\ . \qquad \square$$

In particular, if we choose $f = \langle -,- \rangle : (x : A) \to B(x) \to (\Sigma x : A)B(x)$, Lemma 2.21 gives us a way to deconstruct a goal of equality at a Σ type; there is a term

$$\equiv\text{-}\mathsf{pair} : (p : x \equiv_A y) \to \mathsf{subst}(B,p,u) \equiv_{B(y)} v \to \langle x,u \rangle \equiv_{(\Sigma x:A)B(x)} \langle y,v \rangle$$

such that $\equiv\text{-}\mathsf{pair}(x,x,u,u,\mathsf{refl},\mathsf{refl}) = \mathsf{refl}$. On the other hand, if we have such a term, we can also define cong_2:

Proposition 2.22 The terms cong_2 and $\equiv\text{-}\mathsf{pair}$ are interderivable.

Proof. We have already seen how $\equiv\text{-}\mathsf{pair}$ can be defined as $\equiv\text{-}\mathsf{pair} :\equiv \mathsf{cong}_2(\langle -,- \rangle)$. If we have $\equiv\text{-}\mathsf{pair}$, we can construct $\mathsf{cong}_2(f)$ by packing up the arguments to f in a Σ type and applying the non-dependent cong from Lemma 2.20: we define

$$\mathsf{cong}_2(f,x,y,u,v,p,q) :\equiv \mathsf{cong}(\lambda w.\,f(\mathsf{fst}(w),\mathsf{snd}(w)), \langle x,u \rangle, \langle y,v \rangle, \equiv\text{-}\mathsf{pair}(p,q))\ . \qquad \square$$

Since cong is defined in terms of subst, not the general elimination rule $\mathsf{elim}_{\equiv}$, also cong_2 can be defined using subst and $\equiv$-pair only. This is important in case we want to use cong_2 in the framework of Observational Type Theory, where the general $\mathsf{elim}_{\equiv}$ is not available ($\equiv$-pair, on the other hand, is more or less the definition of equality of dependent pairs in Observational Type Theory).

2.1.7 Propositional types

We call a type *propositional* if it has at most one inhabitant, up to definitional equality. Formally, the type A is propositional if the following rule is admissible:

$$\frac{x, y : A}{x = y : A}$$

This can be compared to the concept of a *mere proposition* or *h-proposition* in Homotopy Type Theory [The Univalent Foundations Program, 2013]. The type A is a mere proposition if

$$(x : A) \to (y : A) \to x \equiv_A y$$

is inhabited. In other words, a mere proposition is a type with at most one inhabitant up to propositional equality. Thus, this property can be internalised in Type Theory, in contrast to the property of being propositional, which lives one level higher. Note that in extensional type theory, a type is propositional if and only if it is an mere proposition by the equality reflection rule. We will not pursue this connection further.

Proposition 2.23

(i) The unit type $\mathbf{1}$ is propositional.

(ii) In extensional Type Theory, the identity type $x \equiv_A y$ is propositional.

(iii) Let $A, B : \mathsf{Set}$. If B is propositional, then so is $A \to B$.

(iv) Let $A : \mathsf{Set}$ and $B : A \to \mathsf{Set}$. If A is propositional, and $B(x)$ is propositional for each $x : A$, then both $(\Sigma x : A)B(x)$ and $(x : A) \to B(x)$ are propositional.

Proof. This all follows from η-rules for the corresponding types, and is straightforward, except possibly the function types in items (iii) and (iv). Let $f, g : A \to B$. By the η-rules for functions, $f = \lambda(x{:}A).\, f(x)$ and $g = \lambda(y{:}A).\, g(y)$ for fresh variables x and y. But B is propositional, and $f(x), g(y) : B$, so $f(x) = g(y)$. Hence by the ξ-rule,

$$f = \lambda(x{:}A).\, f(x) = \lambda(y{:}A).\, g(y) = g \ .$$

In item (iv), we must also ask that A is propositional, so that $x = y$ and thus $B(x) = B(y)$. □

Note that the empty type $\mathbf{0}$ is not propositional, since we have no rule that says that two variables $x, y : \mathbf{0}$ are definitionally equal. In the same way, we cannot expect $(x : A) \to B(x)$ to be propositional for all propositional $B(x)$ if A is non-propositional, even if this seems semantically justified. An alternative, used by Altenkirch [1999] to build a model of extensional Type Theory in an intensional setting, is to introduce a subuniverse Prop of propositional types, together with a "proof-irrelevance" rule

$$\frac{A : \mathsf{Prop} \quad x, y : A}{x = y : A} \ (\textsc{proof-irr})$$

We can safely add $\mathbf{0} : \mathsf{Prop}$ and $(\Pi x : A)B(x) : \mathsf{Prop}$ for $A : \mathsf{Set}$ and $B : A \to \mathsf{Prop}$, since these types will be propositional for instance in the standard set-theoretical model (see Section 5.1). In general, we could consider to at least add the class of Harrop formulas from first-order predicate logic [Harrop, 1960; Troelstra, 1973] to Prop. In particular, this includes the propositional types from Proposition 2.23, which will be propositional in *all* models. Altenkirch proves that the addition of a universe of propositional types does not destroy any nice properties of the theory; it is still decidable, consistent and adequate.

We will find no need to introduce such a universe Prop. Instead, we will mostly be interested in propositional types as a means to improve notation and readability, by introducing "a poor man's subset types": if $P(x)$ is propositional, we will write $\{x : A \mid P(x)\}$ for the dependent sum $\Sigma x : A.P(x)$, and treat $y : \{x : A \mid P(x)\}$ as fst y. If we need to give a term of type $\{x : A \mid P(x)\}$, we will simply give a term $y : A$ and then separately check that there is a term $q : P(y)$ instead of giving the pair $\langle y, q \rangle : \{x : A \mid P(x)\}$. Since $P(y)$ is propositional, any q is as good as any other (they are all the same!), and there is no danger of confusion.

Salvesen and Smith [1988] studied the notion of proper subset types $\{x : A \mid P(x)\}$ in both intensional and extensional type theory. They find that the notion is hard to use in intensional type theory, but usable for $\neg\neg$-stable types P[2] in extensional type theory. In particular, all Harrop formulas are $\neg\neg$-stable, which supports our modest use of subset types in this thesis.

2.2 The dependently typed programming language and proof assistant Agda

Agda [Norell, 2007] is a dependently typed programming language, and, through the Curry-Howard isomorphism, also a proof assistant. The meta-theory of Agda is not very well understood – indeed, one goal of this thesis is to justify the data types that Agda permits – but Agda implements at least Martin-Löf Type Theory (in a Logical Framework formulation) with a tower of universes à la Russel $\mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 \ldots$ and inductive-recursive and inductive-inductive definitions. Section 7.1 is written completely in Agda, and Appendix A contains Agda formalisations of other parts of

[2] That is, types $P : A \to \mathsf{Set}$ for which $(\forall x : A)(\neg\neg P(x) \to P(x))$ is derivable.

the thesis, but knowledge of Agda is not a prerequisite for understanding this thesis. We quickly mention some features of Agda, and how we deal with similar problems in the text:

- Dependent function types are written $(x : \mathsf{A}) \to \mathsf{B}\ x$. Agda also supports implicit arguments, written $\{x : \mathsf{A}\} \to \mathsf{B}\ x$, i.e. the argument $x : \mathsf{A}$ does not need to be specified when the function is applied. This is a very useful feature, but sometimes already the types can be quite cluttered and hard to read, even if the arguments are implicit. In this thesis, we will trust that the reader is able to fill in implicit arguments herself without declaring them as such first – a luxury an actual computer implementation of course cannot afford.

- Data types are declared in Agda using the data keyword, for instance

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

  In this thesis, we will not introduce many data types "by hand"; most data types will be represented by codes in some universe. In an actual implementation, the user could of course be allowed to write data declarations which are then desugared to the underlying codes [Dagand and McBride, 2013].

- Mutual definitions can be introduced by first giving the type of all objects to be defined, then later their definitions, without types:

  ```
  data Even   : ℕ → Set
  data Odd    : ℕ → Set

  data Even where
    ez     : Even zero
    o+1    : {n : ℕ} → Odd n → Even (suc n)

  data Odd where
    e+1    : {n : ℕ} → Even n → Odd (suc n)
  ```

  This way, both inductive-recursive and inductive-inductive definitions are supported in Agda.

- Agda supports dependent pattern matching [Coquand, 1992], instead of using the elimination rules directly. A termination checker checks that all recursive calls are on structurally decreasing arguments. For instance, Agda accepts the following proof that every natural number is either even or odd:

  ```
  evenOdd : (n : ℕ) → Even n + Odd n
  ```

```
evenOdd zero = inl ez
evenOdd (suc n) = [ inr ∘ e+1 , inl ∘ o+1 ] (evenOdd n)
```

We will often use pattern matching notation as an abbreviation for the corresponding eliminator. We emphasise that this is a harmless shorthand, and that we are *not* using the general reduction of pattern matching to eliminators plus Streicher's Axiom K [Goguen et al., 2006].

2.3 Category theory in Type Theory

We will use well-known concepts from category theory without comment. A good introduction can be found in Mac Lane [1998]. However, a few words should perhaps be said about meta-theoretical issues. We are often working in category theory inside Type Theory. When doing so, we usually work in extensional Type Theory. With a little bit of more care, it should be possible to follow Huet and Saïbi [1998] and work with setoids instead (see also Wilander [2012]).

We are also purposefully blurring the distinction between the category of sets and the category generated from the objects of type Set in the Type Theory described in this chapter – who are we to say what the "real" category of sets look like? We only have to be careful that it still has the properties we expect, e.g. that it is complete, locally Cartesian closed, well-pointed, ... (see also Palmgren [2012]).

Chapter 3

A finite axiomatisation of inductive-inductive definitions

Contents

We start by giving some informal examples of inductive-inductive definitions that will also serve as running examples for the rest of the chapter. We then present a finite axiomatisation of a type theory with inductive-inductive definitions, including formation, introduction and elimination rules.

Parts of this chapter have previously been published in the proceedings of CSL 2010 [Nordvall Forsberg and Setzer, 2010] and the Schwichtenberg Festschrift [Nordvall Forsberg and Setzer, 2012].

3.1 Examples of inductive-inductive definitions

In this section, we give some examples of inductive-inductive definitions, starting with the perhaps most important one:

Example 3.1 (Contexts and types) Danielsson [2007] and Chapman [2009] model the syntax of dependent type theory in the theory itself by inductively defining contexts, types (in a given context) and terms (of a given type). To see the inductive-inductive nature of the construction, it is enough to concentrate on contexts and types.

Informally, we have an empty context ε, and if we have any context Γ and a valid type σ in that context, then we can extend the context with a fresh variable $x : \sigma$ to get a new context $\Gamma, x : \sigma$. This is the only way contexts are formed. We end up with the following inductive definition of the set of contexts (with $\Gamma \rhd \sigma$ meaning $\Gamma, x : \sigma$ since we are using de Bruijn indices):

$$\frac{}{\varepsilon : \mathsf{Ctxt}} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma)}{\Gamma \rhd \sigma : \mathsf{Ctxt}}$$

Moving on to types, we have a base type ι (valid in any context) and dependent function types: if σ is a type in context Γ, and τ is a type in $\Gamma, x : \sigma$ (x is the variable from the domain), then $\Pi(\sigma, \tau)$ is a type in the original context. This leads us to the following inductive definition of $\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set}$:

$$\frac{\Gamma : \mathsf{Ctxt}}{\iota_\Gamma : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma) \quad \tau : \mathsf{Ty}(\Gamma \rhd \sigma)}{\Pi_\Gamma(\sigma, \tau) : \mathsf{Ty}(\Gamma)}$$

Note that the definition of Ctxt refers to Ty, so both sets have to be defined simultaneously. Note also how the introduction rule for Π explicitly focuses on a specific constructor in the index of the type of τ. ■

Often, one wishes to define a set A where all elements of A satisfy some property $P : A \to \mathsf{Set}$. If P is inductively defined, one can define A and P simultaneously and achieve that every element of A satisfies P by construction. One example of such a data type is the type of sorted lists:

Example 3.2 (Sorted lists) Let us define a data type consisting of sorted lists (of natural numbers, say). With induction-induction, we can simultaneously define the set SortedList of sorted lists and the predicate $\leq_{\mathrm{L}}$: $(\mathbb{N} \times \mathsf{SortedList}) \to \mathsf{Set}$ with $n \leq_{\mathrm{L}} \ell$ true if n is less than or equal to every element of ℓ.

The empty list is certainly sorted, and if we have a proof p that n is less than or equal to every element of the list ℓ, we can put n in front of ℓ to get a new sorted list $\mathsf{cons}(n, \ell, p)$. Translated into introduction rules, this becomes:

$$\frac{}{\mathsf{nil} : \mathsf{SortedList}} \qquad \frac{n : \mathbb{N} \quad \ell : \mathsf{SortedList} \quad p : n \leq_{\mathrm{L}} \ell}{\mathsf{cons}(n, \ell, p) : \mathsf{SortedList}}$$

For $\leq_{\mathrm{L}}$, we have that every $m : \mathbb{N}$ is trivially smaller than every element of the empty list, and if $m \leq n$ and inductively $m \leq_{\mathrm{L}} \ell$, then $m \leq_{\mathrm{L}} \mathsf{cons}(n, \ell, p)$:

$$\frac{}{\mathsf{triv}_m : m \leq_{\mathrm{L}} \mathsf{nil}} \qquad \frac{q : m \leq n \quad p_{m,\ell} : m \leq_{\mathrm{L}} \ell}{\ll q, p_{m,\ell} \gg : m \leq_{\mathrm{L}} \mathsf{cons}(n, \ell, p)}$$

This makes sense even if the order $\leq$ is not transitive. If it is (as the standard order on the natural numbers is, for example), the argument $p_{m,\ell} : m \leq_{\mathrm{L}} \ell$ can be dropped from the constructor $\ll \cdot \gg$, since we already have $q : m \leq n$ and $p : n \leq_{\mathrm{L}} \ell$, hence by transitivity we must have $m \leq_{\mathrm{L}} \ell$.

Of course, there are also many alternative ways to define such a data type using ordinary induction (or using e.g. induction-recursion, similarly to C. Coquand's definition of fresh lists as reported by Dybjer [2000]). ■

Finally, let us consider an example of more mathematical flavour.

Example 3.3 Recall that an order $<$ on a set A (i.e. a binary relation $< \; : A \to A \to \mathsf{Set}$) is called *dense* if there is a point between any two comparable points, i.e. if $x < y$ then there exists a z in A such that $x < z < y$. The standard order on the rationals is dense, but the standard order on the integers is not.

Given an ordered set $(A,<)$, the *dense completion* $(A^*,<^*)$ of $(A,<)$ is the "least" ordered set in which $(A,<)$ embeds: There is an order-preserving map $\eta : (A,<) \to (A^*,<^*)$, and any order-preserving $f : (A,<) \to (B,\prec)$, where $(B,\prec)$ is a densely ordered set, factors through η in an order-preserving way:

$$\begin{tikzcd}
(A,<) \arrow[r, "\eta"] \arrow[dr, "f"'] & (A^*,<^*) \arrow[d, dashed, "\overline{f}"] \\
 & (B,\prec)
\end{tikzcd}$$

The dense completion of $(A,<)$ can be defined as an inductive-inductive definition. An ordinary inductive definition is not enough, since we need to define the set A^* simultaneously with the order relation $<^*$: $A \to A \to \mathsf{Set}$.

The first constructor of A^* embeds A, while the second adds midpoints:

$$\frac{a : A}{\eta(a) : A^*} \qquad\qquad \frac{x, y : A^* \qquad p : x <^* y}{\mathsf{mid}(x,y,p) : A^*}$$

The order relation $<^*$ is designed to make η order-preserving, and put $\mathsf{mid}(x,y,\_)$ between x and y.

$$\frac{a, b : A \qquad q : a < b}{\eta^{<}(a,b,q) : \eta(a) <^* \eta(b)}$$

$$\frac{x, y : A^* \qquad p : x <^* y}{\mathsf{mid}^{<}_{\mathsf{R}}(x,y,p) : x <^* \mathsf{mid}(x,y,p)} \qquad\qquad \frac{x, y : A^* \qquad p : x <^* y}{\mathsf{mid}^{<}_{\mathsf{L}}(x,y,p) : \mathsf{mid}(x,y,p) <^* y}$$

Notice that this would have been quite hard to express as a recursive definition, as we are not giving $c(x) <^* d(y)$ for all constructors c and d. This concludes the definition of $(A^*,<^*)$. It is clear that $<^*$ is dense:

$$\lambda(x{:}A^*).\,\lambda(y{:}A^*).\,\lambda(p{:}x <^* y).\,\langle \mathsf{mid}(x,y,p), \langle \mathsf{mid}^{<}_{\mathsf{R}}(x,y,p), \mathsf{mid}^{<}_{\mathsf{L}}(x,y,p)\rangle\rangle$$

is a proof of

$$(x, y : A^*) \to x <^* y \to (\Sigma z : A^*)(x < z \times z < y) \ .$$

Furthermore, η is an order-preserving embedding of $(A,<)$ into $(A^*,<^*)$.

We can use the elimination rules for $(A^*,<^*)$ to factor $f : (A,<) \to (B,\prec)$ (with $\prec$ dense, witnessed by $\mathsf{dense}_{\prec}$, say) through η. Let us keep the notation informal, in a pattern-matching style, for now. We want to define $\overline{f} : A^* \to B$, and prove that it is order-preserving. As is typical for inductive-inductive definitions, we need to do this at the same time. Thus, we simultaneously define

$$\begin{aligned} \overline{f} &: A^* \to B \\ \overline{f}^< &: (x, y : A^*) \to x <^* y \to \overline{f}(x) < \overline{f}(y) \end{aligned}$$

For z of the form $z = \eta(a)$, we do not have any choice in how to define $\overline{f}$; if we want f to factor through it, we must have

$$\overline{f}(\eta(a)) = f(a)$$

For z of the form $z = \mathsf{mid}(x, y, p)$, we know the value of $\overline{f}(x)$ and $\overline{f}(y)$ by the induction hypothesis, and we might hope to use the denseness of $<$ to define $\overline{f}(\mathsf{mid}(x, y, p))$; we have a proof $\mathsf{dense}_<$ of

$$\mathsf{dense}_< : (x, y : B) \to x < y \to (\Sigma z : B)(x < z \times z < y)$$

To use this, we need a proof that $\overline{f}(x) < \overline{f}(y)$. But we have a proof $p : x <^* y$, so since we are simultaneously proving that $\overline{f}$ is order-preserving, we can define

$$\overline{f}(\mathsf{mid}(x, y, p)) = \mathsf{fst}(\mathsf{dense}_<(\overline{f}(x), \overline{f}(y), \overline{f}^<(x, y, p))) \ .$$

It remains to define $\overline{f}^< : (x, y : A^*) \to x <^* y \to \overline{f}(x) < \overline{f}(y)$, which we are also allowed to do by structural recursion, this time over $x <^* y$. We have a case for each constructor, and they can be taken care of in the following way, where $f^< : (x, y : A) \to x < y \to f(x) < f(y)$ is the proof that f is order-preserving:

$$\begin{aligned} \overline{f}^<(\_, \_, \eta^<(a, b, q)) &= f^<(a, b, q) \\ \overline{f}^<(\_, \_, \mathsf{mid}^<_{\mathsf{R}}(x, y, p)) &= \mathsf{fst}(\mathsf{snd}(\mathsf{dense}_<(\overline{f}(x), \overline{f}(y), \overline{f}^<(x, y, p)))) \\ \overline{f}^<(\_, \_, \mathsf{mid}^<_{\mathsf{L}}(y, y, p)) &= \mathsf{snd}(\mathsf{snd}(\mathsf{dense}_<(\overline{f}(x), \overline{f}(y), \overline{f}^<(x, y, p)))) \end{aligned}$$

We are using a variant of the (in Type Theory provable) Axiom of Choice when we extract a witness from $\mathsf{dense}_<$. It should also be remarked that $\overline{f}$ is not unique, as one might perhaps expect. A counterexample is given by $A = \{a < b\}$ and B consisting of $a < b$, together with two incomparable chains $a < \ldots < c_{-1} < c_0 < c_1 < \ldots < b$ and $a < \ldots < d_{-1} < d_0 < d_1 < \ldots < b$ between them. The dense completion A^* will consist of one dense chain $\eta(a) < \ldots < e_{-1} < e_0 < e_1 < \ldots < \eta(b)$, but now there are two order-preserving choices for the extension $\overline{\mathsf{id}} : A^* \to B$ of the function $\mathsf{id} : A \hookrightarrow B$: we can either map e_i to c_i or to d_i. ■

Note that these examples strictly speaking refer to extensions of inductive-inductive definitions as presented in this chapter. Example 3.1 in full would be an example of defining a telescope A : Set, $B : A \to$ Set, $C : (x : A) \to B(x) \to$ Set, …inductive-inductively. In Example 3.2, A : Set and $B : (A \times I) \to$ Set for some previously defined set I is defined, and Example 3.3 gives an inductive-inductive definition of A : Set, $B : (A \times A) \to$ Set. In Section 6.2, we explore extensions which capture all these examples in full. For pedagogical reasons, we first treat the simpler case A : Set, $B : A \to$ Set.

3.2 A finite axiomatisation

We now give a finite axiomatisation of a type theory with inductive-inductive definitions. This axiomatisation has been published in Nordvall Forsberg and Setzer [2012]. It differs slightly from the axiomatisation given in Nordvall Forsberg and Setzer [2010], which was not finite. However, the definable sets should be the same for both axiomatisations.

The main idea, following Dybjer and Setzer's axiomatisation of inductive-recursive definitions [Dybjer and Setzer, 1999], is to construct a universe consisting of codes for inductive-inductive definitions, together with a decoding function, which maps a code φ to the domain of the constructor for the inductively defined set represented by φ. We will actually use two universes: one to describe the constructors for the index set A, and one to describe the constructors of the second component $B : A \to \mathsf{Set}$. Just as the constructors for $B : A \to \mathsf{Set}$ can depend on the constructors for the first set A, the codes in the universe $\mathsf{SP}^0_{\mathrm{B}}(\gamma)$ of codes for the second component will depend on codes $\gamma : \mathsf{SP}^0_{\mathrm{A}}$ for the first component.

3.2.1 Dissecting an inductive-inductive definition

We want to formalise and internalise an inductive-inductive definition given by constructors

$$\mathsf{intro}_{\mathrm{A}} : \Phi_{\mathrm{A}}(A, B) \to A$$

and

$$\mathsf{intro}_{\mathrm{B}} : (x : \Phi_{\mathrm{B}}(A, B, \mathsf{intro}_{\mathrm{A}})) \to B(\theta(x))$$

for some $\Phi_{\mathrm{A}}(A, B) : \mathsf{Set}$, $\Phi_{\mathrm{B}}(A, B, \mathsf{intro}_{\mathrm{A}}) : \mathsf{Set}$ and $\theta : \Phi_{\mathrm{B}}(A, B, \mathsf{intro}_{\mathrm{A}}) \to A$. Here, $\theta(x)$ is the *index* of $\mathsf{intro}_{\mathrm{B}}(x)$, i.e. the element $a : A$ such that $\mathsf{intro}_{\mathrm{B}}(x) : B(a)$.

Not all expressions Φ_{A} and Φ_{B} give rise to acceptable inductive-inductive definitions. It is well known, for example, that the theory easily becomes inconsistent if A or B occur in negative positions in Φ_{A} or Φ_{B} respectively. Thus, we restrict our attention to a class of strictly positive functors.

These are based on the following analysis of what kind of premises can occur in a definition. A premise is either *inductive* or *non-inductive*. A non-inductive premise consists of a previously constructed set K, on which later premises can depend. An inductive premise is inductive in A or B. If it is inductive in A, it is of the form $K \to A$ for some previously constructed set K. Premises inductive in B are of the form $(x : K) \to B(i(x))$ for some $i : K \to A$.

If $K = \mathbf{1}$, we have the special case of a single inductive premise. In the case of B-inductive arguments, the choice of $i : \mathbf{1} \to A$ is then just a choice of a single element $a = i(\star) : A$ so that the premise is of the form $B(a)$. This is called an *ordinary* inductive premise, with the general case called a *generalised* inductive premise.

3.2.2 Dybjer and Setzer's axiomatisation of inductive-recursive definitions

To get used to the style of axiomatisation we are going to use, and the idea of using universes of codes to represent inductively defined types, we recall Dybjer and

Setzer's [1999] axiomatisation of a type theory with inductive-recursive definitions. Even though inductive-recursive definitions are proof-theoretically much stronger than inductive-inductive definitions (see Section 5.3 for a partial result), they admit a simpler axiomatisation. This is not as paradoxical as it sounds; it is simply the case that we can give a more uniform and straight-forward description of the universe of codes used to describe inductive-recursive types. The introduction and elimination rules, which is where the formal power comes from, are equally simple to state for both theories, but have very different consequences.

An inductive-recursive definition consists of an inductively defined set U, and a recursively defined function $T : U \to D$ for some (possibly large) type D. That U is inductively defined means that it is given by a constructor

$$\mathsf{intro} : \Phi(U, T) \to U$$

and that T is recursively defined means that the value of T on a canonical element $\mathsf{intro}(\vec{x})$ of U is given in terms of the value of T on the subterms of $\vec{x}$.

Dybjer and Setzer's idea was to describe the domain $\Phi(U,T)$ of intro – but for arbitrary $U : \mathsf{Set}$, $T : U \to D$. This domain is basically a list (telescope) of arguments, as in Section 3.2.1, where later arguments can depend on earlier ones. If the argument is non-inductive, this dependency is direct, whereas for an inductive argument $u : U$, we can only depend on $T(u) : D$. This intuitively makes sense, as we do know what the elements of D are, but not the elements of U – we are in the middle of the process of defining them!

Dybjer and Setzer made the above observations formal by defining a large type $\mathsf{IR}\,D$ of codes for $\Phi(U,T)$, together with decoding functions

$$\begin{aligned}\mathsf{Arg}_{\mathsf{IR}}(\gamma) &: (U : \mathsf{Set}) \to (T : U \to D) \to \mathsf{Set}\\ \mathsf{Fun}_{\mathsf{IR}}(\gamma) &: (U : \mathsf{Set}) \to (T : U \to D) \to \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T) \to D\end{aligned}$$

for each code $\gamma : \mathsf{IR}\,D$. Here $\mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T)$ should be thought of as the domain of the constructor $\mathsf{intro}_\gamma : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T) \to U$, and $\mathsf{Fun}_{\mathsf{IR}}(\gamma, U, T, \vec{x})$ as the value of $T(\mathsf{intro}_\gamma(\vec{x}))$. The type $\mathsf{IR}\,D$ needs to be large, since it is referring to to arbitrary sets. The codes in $\mathsf{IR}\,D$, and their decodings, are inductively defined by the following clauses:

The code $\iota(d)$ represents a trivial constructor $\mathsf{intro}_{\iota(d)} : \mathbf{1} \to U$ (a base case):

$$\frac{d : D}{\iota(d) : \mathsf{IR}\,D} \qquad \mathsf{Arg}_{\mathsf{IR}}(\iota(d), U, T) = \mathbf{1}$$

The code $\sigma(A, f)$ represents a noninductive argument $a : A$, followed by the rest of the arguments, which are represented by $f(a)$. The name σ stands for the Σ type, which is used in its decoding:

$$\frac{A : \mathsf{Set} \qquad f : A \to \mathsf{IR}\,D}{\sigma(A, f) : \mathsf{IR}\,D} \qquad \mathsf{Arg}_{\mathsf{IR}}(\sigma(A, f), U, T) = (\Sigma a : A)\mathsf{Arg}_{\mathsf{IR}}(f(a), U, T)$$

Note that the remaining arguments $f(a)$ can depend on a.

The code $\delta(A, F)$ represents an inductive argument $g : A \to U$, followed by the rest of the arguments, which are represented by $F(T \circ g)$. The name δ stands for "dependent Σ":

$$\frac{A : \mathsf{Set} \qquad F : (A \to D) \to \mathsf{IR}\,D}{\delta(A, F) : \mathsf{IR}\,D}$$

$$\mathsf{Arg}_{\mathsf{IR}}(\delta(A, F), U, T) = (\Sigma g : A \to U)\mathsf{Arg}_{\mathsf{IR}}(F(T \circ g), U, T)$$

Note that the rest of the arguments, which are represented by $F(T \circ g)$, do not depend on the argument $g : A \to U$ directly, but only on $T \circ g$.[1]

For completeness, we also give the definition of $\mathsf{Fun}_{\mathsf{IR}}$, although the details will not interest us very much until Section 6.1.

$$\begin{aligned}
\mathsf{Fun}_{\mathsf{IR}}(\iota(d), U, T, \star) &= d \\
\mathsf{Fun}_{\mathsf{IR}}(\sigma(A, f), U, T, \langle a, x \rangle) &= \mathsf{Fun}_{\mathsf{IR}}(f(a), U, T, x) \\
\mathsf{Fun}_{\mathsf{IR}}(\delta(A, F), U, T, \langle g, x \rangle) &= \mathsf{Fun}_{\mathsf{IR}}(F(T \circ g), U, T, x)
\end{aligned}$$

The principle of inductive-recursive definitions now states that there is a family (U_γ, T_γ) "closed under" $\mathsf{Arg}_{\mathsf{IR}}(\gamma)$ and $\mathsf{Fun}_{\mathsf{IR}}(\gamma)$ for each code $\gamma : \mathsf{IR}\,D$. Formally, this is expressed using the following rules:

Axioms 3.4 (Rules for inductive-recursive definitions) Formation rules:

$$\frac{D \text{ type} \qquad \gamma : \mathsf{IR}\,D}{U_\gamma : \mathsf{Set}} \qquad\qquad \frac{D \text{ type} \qquad \gamma : \mathsf{IR}\,D}{T_\gamma : U_\gamma \to \mathsf{Set}}$$

For the rest of the rules, we suppress the premises D type and $\gamma : \mathsf{IR}\,D$.

Introduction rule for U_γ:

$$\frac{a : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U_\gamma, T_\gamma)}{\mathsf{intro}_\gamma(a) : U_\gamma}$$

Computation rule for T_γ:

$$T_\gamma(\mathsf{intro}_\gamma(a)) = \mathsf{Fun}_{\mathsf{IR}}(\gamma, U_\gamma, T_\gamma, a) \qquad ∎$$

We do not assume that these axioms are part of our type theory in general. Let us now look at an example of how the theory is meant to be used.

Example 3.5 (A universe closed under W-types) The code

$$\gamma_{\mathsf{W}} := \delta(\mathbf{1}, \lambda X.\, \delta(X(\star), \lambda Y.\, \iota(\mathsf{W}(x : X(\star))Y(x))))$$

describes a universe (U, T) closed under W-types. If we decode it, we get

$$\mathsf{Arg}_{\mathsf{IR}}(\gamma_{\mathsf{W}}, U, T) = (\Sigma a : \mathbf{1} \to U)(\Sigma b : T(a(\star)) \to U)\mathbf{1} \cong (\Sigma a : U)(T(a) \to U)$$

[1] For inductive-inductive definitions, we have no recursively defined function $T : U \to D$, and hence no such general dependency on inductive arguments.

so that the constructor of the universe has type isomorphic to

$$\mathsf{intro}_{\gamma_{\mathsf{W}}} : (a : U) \to (b : T(a) \to U) \to U$$

after uncurrying, and we have

$$T(\mathsf{intro}_{\gamma_{\mathsf{W}}}(a, b)) = \mathsf{W}(x : T(a))T(b(x)) \ .$$

This shows that inductive-recursive definitions can be used for proof-theoretically strong constructions [Setzer, 1998]. Dybjer and Setzer go considerable further and show that e.g. Palmgren's superuniverse [Palmgren, 1998] and Setzer's external Mahlo universe [Setzer, 2008] are subsumed by the theory of (indexed) inductive-recursive definitions. ■

When it comes to elimination rules for inductive-recursive definitions, Martin-Löf [1972] writes about a universe V:

> It is not natural although possible to add the principle of (transfinite) induction over V, expressing the idea that V is the least type which is closed with respect to the above inductive clauses, because we want to keep our universe open so as to be free to throw new types into it or require it to be closed with respect to new type forming operations.

These considerations are not so important with the principle of inductive-recursive definitions at our disposal, as we can just construct a second universe, should we need it to contain more types. Furthermore, if we use inductive-recursive definitions to construct structures that are not universes (e.g. balanced binary trees [Ek et al., 2009]), it is crucial to have an elimination principle for those structures. Hence Dybjer and Setzer also define elimination rules for inductive-recursively defined sets. They define a set of induction hypothesis[2] for a given element $x : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T)$

$$\frac{\gamma : \mathsf{IR}\,D \quad \overset{U\,:\,\mathsf{Set}}{T : U \to D} \quad P : U \to \mathsf{Set} \quad x : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T)}{\mathsf{IH}_{\mathsf{IR}}(\gamma, U, T, P, x) : \mathsf{Set}}$$

by induction over γ, together with a function $\mathsf{map}_{\mathsf{IH}}$ which takes care of the recursive calls:

$$\frac{\ldots \quad g : (x : U) \to P(x) \quad x : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T)}{\mathsf{map}_{\mathsf{IH}}(\gamma, U, T, P, g, x) : \mathsf{IH}_{\mathsf{IR}}(\gamma, U, T, p, x)}$$

Given these operations, we can define the elimination rule for U_γ to be

$$\frac{x : U_\gamma \vdash P(x)\ \mathsf{type} \quad g : (x : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U_\gamma, T_\gamma)) \to \mathsf{IH}_{\mathsf{IR}}(\gamma, U_\gamma, T_\gamma, P, x) \to P(\mathsf{intro}_\gamma(x)) \quad u : U_\gamma}{\mathsf{elim}_\gamma(P, g, u) : P(x)}$$

[2] Actually, since the codomain D of the recursive function T can be large, it makes sense for the elimination rules to support large elimination as well. This forces also the collection of induction hypothesis to be large, and complicates the construction slightly [Dybjer and Setzer, 2006, Section 5.4]. For our purposes, small elimination will be enough.

with computation rule

$$\mathsf{elim}_\gamma(P, g, \mathsf{intro}_\gamma(x)) = g(x, \mathsf{map}_{\mathsf{IH}}(\gamma, U_\gamma, T_\gamma, P, \mathsf{elim}_\gamma(P, g), x) : P(\mathsf{intro}_\gamma(x)) \ .$$

Another option, explored in Dybjer and Setzer [2003], and whose analogue for inductive-inductive definitions we will pursue in Chapter 4, starts with the observation that $\mathsf{Arg}_{\mathsf{IR}}(\gamma)$ together with $\mathsf{Fun}_{\mathsf{IR}}(\gamma)$ can be extended to an endofunctor on a category $\mathsf{Fam}\,\mathbb{D}$:

Definition 3.6 Let $\mathbb{D}$ be a category. The category $\mathsf{Fam}\,\mathbb{D}$ of families of objects of $\mathbb{D}$ has as objects pairs (A, B), where A is a set and $B : A \to \mathbb{D}$ is an A-indexed family of objects of $\mathbb{D}$. A morphism from (A, B) to (A', B') is a pair (f, g) consisting of a function $f : A \to A'$ and a natural transformation $g : B \to B' \circ f$, i.e. $g : (x : A) \to B(x) \to B'(f(x))$. ◼

Given a (possibly large) type D, we can regard it as a discrete category. Note that in this case, a morphism from (A, B) to (A', B') is just a function $f : A \to A'$ such that $B = B' \circ f : A \to D$ since the only morphisms in the category D are identity morphisms. The operations $\mathsf{Arg}_{\mathsf{IR}}(\gamma)$ and $\mathsf{Fun}_{\mathsf{IR}}(\gamma)$ gives rise to an endofunctor on $\mathsf{Fam}\,D$ which maps (U, T) to the family $(\mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T), \mathsf{Fun}_{\mathsf{IR}}(\gamma, U, T))$. To extend this to an action on morphisms, Dybjer and Setzer use extensional equality in an essential way.

Now that every code γ in $\mathsf{IR}\,D$ gives rise to a functor, we can use the machinery of initial algebra semantics [Goguen et al., 1977] to express elimination rules for U_γ. Dybjer and Setzer [2003] show that initiality of $(U_\gamma, T_\gamma, \mathsf{intro}_\gamma)$ is equivalent to the elimination rules we have formulated above.

3.2.3 The axiomatisation of inductive-inductive definitions

We now give the formal rules for inductive-inductive definitions. These consists of a set of rules for the universe SP_{A} of descriptions of the set A and its decoding function $\mathsf{Arg}_{\mathsf{A}}$, a set of rules for the universe SP_{B} and its decoding function $\mathsf{Arg}_{\mathsf{B}}$, and formation and introduction rules for $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$ defined inductive-inductively by a pair of codes $\gamma_{\mathsf{A}} : \mathsf{SP}_{\mathsf{A}}$, $\gamma_{\mathsf{B}} : \mathsf{SP}_{\mathsf{B}}(\gamma_{\mathsf{A}})$. The elimination rules will be dealt with in Section 3.2.5. The concepts involved in the axiomatisation are summarised in Table 3.1.

We first define the universe SP_{A} of codes for $A : \mathsf{Set}$ and its decoding function $\mathsf{Arg}_{\mathsf{A}}$ in Section 3.2.3.1. Important will be the concept of "referable" elements in the process of constructing a code $\gamma_{\mathsf{A}} : \mathsf{SP}_{\mathsf{A}}$. For instance, after an inductive argument $a : X$, all later arguments can refer to a. We collect all such referable elements in a set X_{ref}, together with a function $\mathsf{rep}_{\mathsf{X}} : X_{\mathrm{ref}} \to X$ which makes X_{ref} a "subset" of X.

When we get to codes for the second family $B : A \to \mathsf{Set}$, we see that there is a slight complication: since we want the constructor for B to be able to refer to the constructor $\mathsf{intro}_{\mathsf{A}} : \mathsf{Arg}_{\mathsf{A}}(\gamma_{\mathsf{A}}, A, B) \to A$, we must also make it possible to refer to elements of the form $\mathsf{intro}_{\mathsf{A}}(\vec{x})$ or even $\mathsf{intro}_{\mathsf{A}}(\mathsf{intro}_{\mathsf{A}}(\vec{x}), \vec{y})$ etc. The necessary machinery is developed in Section 3.2.3.2, where a set $\mathsf{A\text{-}Term}(\gamma_{\mathsf{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$, consisting of terms constructed from X_{ref}, Y_{ref} and $\mathsf{intro}_{\mathsf{A}}$, is defined. We also define a function $\overline{\mathsf{rep}_{\mathsf{A}}}$ which makes $\mathsf{A\text{-}Term}(\gamma_{\mathsf{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ a "subset" of X.

Table 3.1: Concepts involved in the axiomatisation.

Name	Meaning	Section
SP_{A}	universe of codes γ_{A} for first set A	3.2.3.1
$\mathsf{Arg}_{\mathrm{A}}$	decoding function	
X_{ref}	current referable terms from first set	
$\mathsf{rep}_{\mathrm{X}}$	function witnessing $X_{\mathrm{ref}} \subseteq X$	
Y_{ref}	current referable terms from second family	3.2.3.2
$\mathsf{rep}_{\mathrm{index}}$, $\mathsf{rep}_{\mathrm{Y}}$	functions witnessing $(x : Y_{\mathrm{ref}}) \subseteq Y(\mathsf{rep}_{\mathrm{index}}(x))$	
A-Term	terms built from referable terms and constructor	
$\overline{\mathsf{rep}_{\mathrm{A}}}$	function witnessing $\text{A-Term}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}) \subseteq X$	
$\mathsf{SP}_{\mathrm{B}}(\gamma_{\mathrm{A}})$	universe of codes γ_{B} for second family $B : A \to \mathsf{Set}$	3.2.3.3
$\mathsf{Arg}_{\mathrm{B}}$	decoding function	
$\mathsf{Index}_{\mathrm{B}}$	targeted index function	
$A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$, $B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$	the inductive-inductively defined family	3.2.3.4
$\mathsf{intro}_{\mathrm{A}}$, $\mathsf{intro}_{\mathrm{B}}$	constructors	

In Section 3.2.3.3, we then make use of the machinery developed in Section 3.2.3.2 to define a universe SP_{B} of codes for $B : A \to \mathsf{Set}$, together with a decoding function $\mathsf{Arg}_{\mathrm{B}}$. We also define a function $\mathsf{Index}_{\mathrm{B}}$ which we intend to pick out the index of the constructed element, i.e. given $x : \mathsf{Arg}_{\mathrm{B}}(\gamma_{\mathrm{B}}, A, B)$, the function $\mathsf{Index}_{\mathrm{B}}$ picks out $a = \mathsf{Index}_{\mathrm{B}}(x) : A$ such that $\mathsf{intro}_{\mathrm{B}}(x) : B(a)$.

Finally, in Section 3.2.3.4, the formation and introduction rules are introduced.

3.2.3.1 The universe $\mathsf{SP}^0_{\mathrm{A}}$ of descriptions of A

We introduce the universe of codes for the index set with the formation rule

$$\frac{X_{\mathrm{ref}} : \mathsf{Set}}{\mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})\ \mathsf{type}}$$

The set X_{ref} should be thought of as the elements of A that we can refer to in the code that we are defining. To start with, we cannot refer to any elements in A, and so we define $\mathsf{SP}^0_{\mathrm{A}} := \mathsf{SP}_{\mathrm{A}}(\mathbf{0})$. After introducing an inductive argument $a : A$, we can refer to a in later arguments, so that X_{ref} will be extended to include a as well for the construction of the rest of the code.

The introduction rules for SP_{A} reflects the informal discussion in Section 3.2.1. The rules are as follows (we suppress the global premise $X_{\mathrm{ref}} : \mathsf{Set}$):

The code nil represents a trivial constructor $c : \mathbf{1} \to A$ (a base case):

$$\frac{}{\mathsf{nil} : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

The code $\mathsf{non\text{-}ind}(K,\gamma)$ represents a non-inductive argument $x : K$, with the rest of the arguments given by $\gamma(x)$:

$$\frac{K : \mathsf{Set} \qquad \gamma : K \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}{\mathsf{non\text{-}ind}(K,\gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

The code $\mathsf{A\text{-}ind}(K,\gamma)$ represents an inductive argument of type $K \to A$, with the rest of the arguments given by γ:

$$\frac{K : \mathsf{Set} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K)}{\mathsf{A\text{-}ind}(K,\gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

Notice that $\gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K)$, so that the remaining arguments can refer to more elements in A (namely those introduced by the inductive argument).

Finally, the code $\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma)$ represents an inductive argument of type $(x : K) \to B(i(x))$, where the index $i(x)$ is determined by h_{index}, and the rest of the arguments are given by γ:

$$\frac{K : \mathsf{Set} \qquad h_{\mathrm{index}} : K \to X_{\mathrm{ref}} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}{\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

Example 3.7 Returning to the contexts and types of Example 3.1, the constructor

$$\rhd \; : \; \big((\Sigma\Gamma : \mathsf{Ctxt})\mathsf{Ty}(\Gamma)\big) \to \mathsf{Ctxt}$$

is represented by the code

$$\gamma_{\rhd} = \mathsf{A\text{-}ind}(\mathbf{1}, \mathsf{B\text{-}ind}(\mathbf{1}, \lambda(\star : \mathbf{1}).\widehat{\Gamma}, \mathsf{nil})) \;\;,$$

where $\widehat{\Gamma} = \mathsf{inr}(\star)$ is the representation of Γ in $X_{\mathrm{ref}} = \mathbf{0} + \mathbf{1}$. ■

We now define the decoding function $\mathsf{Arg}_{\mathrm{A}}$, which maps a code to the domain of the constructor it represents. In addition to a set X_{ref} and a code $\gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})$, $\mathsf{Arg}_{\mathrm{A}}$ will take a set X and a family $Y : X \to \mathsf{Set}$ as arguments to use as A and B in the inductive arguments. These will later be instantiated by the sets defined inductive-inductively ("tying the knot"). We also require a function $\mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X$ which we think of as mapping a "referable" element to the element it represents in X. Thus, via $\mathsf{rep}_{\mathrm{X}}$, we can see X_{ref} as a "subset" of X. $\mathsf{Arg}_{\mathrm{A}}$ has the following formation rule:

$$\frac{X_{\mathrm{ref}} : \mathsf{Set} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}}) \qquad X : \mathsf{Set} \qquad Y : X \to \mathsf{Set} \qquad \mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X}{\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}}) : \mathsf{Set}}$$

Notice that if $\gamma : \mathsf{SP}^0_{\mathrm{A}}$, i.e. if $X_{\mathrm{ref}} = \mathbf{0}$, then we can choose $\mathsf{rep}_{\mathrm{X}} = !_X : \mathbf{0} \to X$ (indeed, extensionally, this is the only choice), so that we can define

$$\mathsf{Arg}^0_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}} \to (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to \mathsf{Set}$$

by $\mathsf{Arg}^0_{\mathrm{A}}(\gamma, X, Y) = \mathsf{Arg}_{\mathrm{A}}(\mathbf{0}, \gamma, X, Y, !_X)$.

The definition of Arg_A follows the informal description of what the different codes represent above[3]:

$$\begin{aligned}
\mathsf{Arg}_\mathsf{A}(\_,\mathsf{nil},\_,\_,\_) &= \mathbf{1}\\
\mathsf{Arg}_\mathsf{A}(\_,\mathsf{non\text{-}ind}(K,\gamma),\_,\_,\_) &= (\Sigma x : K)\mathsf{Arg}_\mathsf{A}(\_,\gamma(x),\_,\_,\_)\\
\mathsf{Arg}_\mathsf{A}(X_\mathrm{ref},\mathsf{A\text{-}ind}(K,\gamma),X,\_,\mathsf{rep}_\mathsf{X}) &=\\
&(\Sigma j : K \to X)\mathsf{Arg}_\mathsf{A}(X_\mathrm{ref}+K,\gamma,\_,\_,[\mathsf{rep}_\mathsf{X},j])\\
\mathsf{Arg}_\mathsf{A}(\_,\mathsf{B\text{-}ind}(K,h_\mathsf{index},\gamma),\_,Y,\mathsf{rep}_\mathsf{X}) &=\\
&((x : K) \to Y((\mathsf{rep}_\mathsf{X} \circ h_\mathsf{index})(x))) \times \mathsf{Arg}_\mathsf{A}(\_,\gamma,\_,\_,\_)
\end{aligned}$$

Example 3.8 Recall the code $\gamma_\triangleright = \mathsf{A\text{-}ind}(\mathbf{1},\mathsf{B\text{-}ind}(\mathbf{1},\lambda(\star:\mathbf{1}).\,\mathsf{inr}(\star),\mathsf{nil}))$ for the constructor $\triangleright : \big((\Sigma\Gamma : \mathsf{Ctxt})\mathsf{Ty}(\Gamma)\big) \to \mathsf{Ctxt}$. We have

$$\mathsf{Arg}^0_\mathsf{A}(\gamma_\triangleright,\mathsf{Ctxt},\mathsf{Ty}) = (\Sigma\Gamma : \mathbf{1} \to \mathsf{Ctxt})(\mathbf{1} \to \mathsf{Ty}(\Gamma(\star))) \times \mathbf{1}$$

which, thanks to the η-rules for $\mathbf{1}$, Σ and $\to$, is isomorphic to the domain of $\triangleright$. ■

3.2.3.2 Towards descriptions of B

As we have seen in Example 3.1, it is important that the constructor $\mathsf{intro}_\mathsf{B}$ for the second set $B : A \to \mathsf{Set}$ can refer to the constructor $\mathsf{intro}_\mathsf{A}$ for the first set A. This means that inductive arguments might be of type $B(\mathsf{intro}_\mathsf{A}(\overline{a}))$ for some $\overline{a} : \mathsf{Arg}^0_\mathsf{A}(\gamma_\mathsf{A}, A, B)$ or even $B(\mathsf{intro}_\mathsf{A}(\ldots\mathsf{intro}_\mathsf{A}\ldots(\overline{a})))$ for some $\overline{a} : \mathsf{Arg}^0_\mathsf{A}(\gamma_\mathsf{A},\ldots\mathsf{Arg}^0_\mathsf{A}(\gamma_\mathsf{A}, A, B)\ldots, B')$. Thus, we need to be able to represent such indices in the descriptions of the constructor $\mathsf{intro}_\mathsf{B}$.

It is no longer enough to only keep track of the referable elements X_ref of X – we need to be able to refer to elements of $B(x)$ as well, since they could be used as arguments to $\mathsf{intro}_\mathsf{A}$. How can we represent elements of a family $Y(x)$, where $Y : X \to \mathsf{Set}$?

- The most direct option is to use another family $Y'_\mathrm{ref} : X_\mathrm{ref} \to \mathsf{Set}$. Where $y : Y'_\mathrm{ref}(x)$ might represent $b : Y(a)$ for some $x : X_\mathrm{ref}$ representing $a : X$. However, this quickly becomes unwieldy, since e.g. enlarging X_ref means modifying Y'_ref as well.
- Instead, we take a more "fibred" approach and represent elements of the family Y by a set Y_ref, together with functions $\mathsf{rep}_\mathsf{index} : Y_\mathrm{ref} \to X$ and $\mathsf{rep}_\mathsf{Y} : (x : Y_\mathrm{ref}) \to Y(\mathsf{rep}_\mathsf{index}(x))$; the function $\mathsf{rep}_\mathsf{index}$ gives the index of the represented element, and rep_Y the actual element. There is no need to factor $\mathsf{rep}_\mathsf{index}$ through X_ref and rep_X (in fact, it would make things more complicated).

We want to represent elements in $\mathsf{Arg}^0_\mathsf{A}(\gamma_\mathsf{A}, X, Y)$. We claim that the elements in $\mathsf{Arg}^0_\mathsf{A}(\gamma_\mathsf{A}, X_\mathrm{ref} + Y_\mathrm{ref}, [\lambda x.\,\mathbf{0}, \lambda x.\,\mathbf{1}])$ are suitable for this purpose. To see this, first observe that we can define functions

$$f : X_\mathrm{ref} + Y_\mathrm{ref} \to X \quad ,$$

[3] For readability, we have replaced arguments which are simply passed on with "_" in the recursive call, and likewise on the left hand side if the argument is not used otherwise. This is different from Agda's use of "_".

$$g : (x : X_{\mathrm{ref}} + Y_{\mathrm{ref}}) \to [\lambda x.\mathbf{0}, \lambda x.\mathbf{1}](x) \to Y(f(x))$$

by $f = [\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}]$ and $g = [\lambda x.!, \lambda x.\lambda\star.\mathsf{rep}_{\mathrm{Y}}(x)]$. These are morphisms between families of sets in the sense of Definition 3.6. Then, we can lift these functions to a function

$$\mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, f, g) : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}} + Y_{\mathrm{ref}}, [\lambda x.\mathbf{0}, \lambda x.\mathbf{1}]) \to \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y)$$

by observing that $\mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}})$ is functorial:

Lemma 3.9 For each $\gamma : \mathsf{SP}^0_{\mathrm{A}}$, $\mathsf{Arg}^0_{\mathrm{A}}(\gamma)$ extends to a functor from families of sets to sets, i.e. given $f : X \to X'$ and $g : (x : X) \to Y(x) \to Y'(f(x))$, one can define $\mathsf{Arg}^0_{\mathrm{A}}(\gamma, f, g) : \mathsf{Arg}^0_{\mathrm{A}}(\gamma, X, Y) \to \mathsf{Arg}^0_{\mathrm{A}}(\gamma, X', Y')$.

Remark. In extensional type theory, one can also prove that $\mathsf{Arg}^0_{\mathrm{A}}(\gamma, f, g)$ actually is a functor, i.e. that identities and compositions are preserved, but that will not be needed for the current development.

Proof. This is straightforward in extensional type theory. In intensional type theory without propositional identity types, we have to be more careful. We define the function $\mathsf{Arg}^0_{\mathrm{A}}(\gamma, f, g)$ by induction over γ. In order to do this, we need to refer inductively to the case when X_{ref} is no longer $\mathbf{0}$. Hence, we need to consider the more general case where X, Y, X', Y', f and g have types as above, and $X_{\mathrm{ref}},$: Set, $\mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X$, $\mathsf{rep}'_{\mathrm{X}} : X_{\mathrm{ref}} \to X'$. One expects the equality $f(\mathsf{rep}_{\mathrm{X}}(x)) = \mathsf{rep}'_{\mathrm{X}}(x)$ to hold for all $x : X_{\mathrm{ref}}$. In order to avoid the use of identity types, we state this in a form of Leibniz equality, specialised to the instance we actually need; we ask also for a term

$$p : (x : X_{\mathrm{ref}}) \to Y'(f(\mathsf{rep}_{\mathrm{X}}(x))) \to Y'(\mathsf{rep}'_{\mathrm{X}}(x)) \ .$$

If $X_{\mathrm{ref}} = \mathbf{0}$, we can trivially use $p =!_{\lambda x.\,Y'(f(\mathsf{rep}_{\mathrm{X}}(x))) \to Y'(\mathsf{rep}'_{\mathrm{X}}(x))}$. We define

$$\mathsf{Arg}_{\mathrm{A}}(\gamma, f, g, p) : \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}}) \to \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X', Y', \mathsf{rep}'_{\mathrm{X}})$$

by induction over γ:

$$\begin{aligned}
\mathsf{Arg}_{\mathrm{A}}(\mathsf{nil}, f, g, p, \star) &= \star \\
\mathsf{Arg}_{\mathrm{A}}(\mathsf{non\text{-}ind}(K, \gamma), f, g, p, \langle k, y\rangle) &= \langle k, \mathsf{Arg}_{\mathrm{A}}(\gamma(k), f, g, p, y)\rangle \\
\mathsf{Arg}_{\mathrm{A}}(\mathsf{A\text{-}ind}(K, \gamma), f, g, p, \langle j, y\rangle) &= \langle f \circ j, \mathsf{Arg}_{\mathrm{A}}(\gamma, f, g, [p, \lambda x.\mathsf{id}], y)\rangle \\
\mathsf{Arg}_{\mathrm{A}}(\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma), f, g, p, \langle j, y\rangle) &= \\
\langle \lambda k.\,p(h_{\mathrm{index}}(k), g(\mathsf{rep}_{\mathrm{X}}(h_{\mathrm{index}}(k)), j(k))), &\mathsf{Arg}_{\mathrm{A}}(\gamma, f, g, p, y)\rangle
\end{aligned}$$

Notice the use of the specialised Leibniz equality p in the last line. Finally, we can define $\mathsf{Arg}^0_{\mathrm{A}}(\gamma, f, g) : \mathsf{Arg}^0_{\mathrm{A}}(\gamma, A, B) \to \mathsf{Arg}^0_{\mathrm{A}}(\gamma, A', B')$ by

$$\mathsf{Arg}^0_{\mathrm{A}}(\gamma, f, g) := \mathsf{Arg}_{\mathrm{A}}(\gamma, f, g, !) \ . \qquad \square$$

Recall that we want to use Lemma 3.9 to represent elements in $\mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y)$ by elements in $\mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}} + Y_{\mathrm{ref}}, [\lambda x.\,\mathbf{0}, \lambda x.\,\mathbf{1}])$. We can actually do better, and represent arbitrarily terms built from elements in X and Y with the use of a constructor $\mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y) \to X$. For this, define the set $\mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ of terms "built from $\mathsf{intro}_{\mathrm{A}}$, X_{ref} and Y_{ref}" with introduction rules

$$\frac{x : X_{\mathrm{ref}}}{\mathsf{a}_{\mathrm{ref}}(x) : \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})}$$

$$\frac{x : Y_{\mathrm{ref}}}{\mathsf{b}_{\mathrm{ref}}(x) : \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})}$$

$$\frac{x : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}), \mathsf{B\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}))}{\mathsf{arg}(x) : \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})}$$

Here, $\mathsf{B\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}) : \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}) \to \mathsf{Set}$ is defined by

$$\begin{array}{lcl}
\mathsf{B\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \mathsf{a}_{\mathrm{ref}}(x)) & = & \mathbf{0} \\
\mathsf{B\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \mathsf{b}_{\mathrm{ref}}(x)) & = & \mathbf{1} \\
\mathsf{B\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \mathsf{arg}(x)) & = & \mathbf{0}
\end{array}$$

Note that this is formally an inductive-recursive definition. The intuition behind the definition of B-Term is that all elements of Y we know are represented in Y_{ref}, and only in Y_{ref}.

All elements in $\mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ represents elements in X, given that we have a function $\mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y) \to X$ and the elements of X_{ref} and Y_{ref} represents elements of X and Y respectively (i.e. we have $\mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X$, $\mathsf{rep}_{\mathrm{index}} : Y_{\mathrm{ref}} \to X$ and $\mathsf{rep}_{\mathrm{Y}} : (x : Y_{\mathrm{ref}}) \to Y(\mathsf{rep}_{\mathrm{index}}(x))$). We think of X_{ref} as containing elements of X constructed without $\mathsf{intro}_{\mathrm{A}}$, and of $\mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ as containing elements of X constructed with an arbitrary number of applications of $\mathsf{intro}_{\mathrm{A}}$ (possibly zero). Formally, we can simultaneously define the following two functions:

$$\frac{\gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}} \qquad \mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y) \to X \qquad \begin{array}{r} \mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X \\ \mathsf{rep}_{\mathrm{index}} : Y_{\mathrm{ref}} \to X \\ \mathsf{rep}_{\mathrm{Y}} : (x : Y_{\mathrm{ref}}) \to Y(\mathsf{rep}_{\mathrm{index}}(x)) \end{array}}{\begin{array}{c} \overline{\mathsf{rep}_{\mathrm{A}}}(\ldots) : \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}) \to X \\ \overline{\mathsf{rep}_{\mathrm{B}}}(\ldots) : (x : \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}})) \to \mathsf{B\text{-}Term}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, x) \to Y(\overline{\mathsf{rep}_{\mathrm{A}}}(\ldots, x)) \end{array}}$$

The definition of $\overline{\mathsf{rep}_{\mathrm{A}}}$ is straightforward. The interesting case is $\mathsf{arg}(x)$, where we make use of the constructor $\mathsf{intro}_{\mathrm{A}}$, the functoriality of $\mathsf{Arg}^0_{\mathrm{A}}$ and the mutually defined $\overline{\mathsf{rep}_{\mathrm{B}}}$:

$$\begin{array}{l}
\overline{\mathsf{rep}_{\mathrm{A}}}(\gamma_{\mathrm{A}}, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{a}_{\mathrm{ref}}(x)) = \mathsf{rep}_{\mathrm{X}}(x) \\
\overline{\mathsf{rep}_{\mathrm{A}}}(\gamma_{\mathrm{A}}, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{b}_{\mathrm{ref}}(x)) = \mathsf{rep}_{\mathrm{index}}(x) \\
\overline{\mathsf{rep}_{\mathrm{A}}}(\gamma_{\mathrm{A}}, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{arg}(x)) = \\
\qquad\qquad \mathsf{intro}_{\mathrm{A}}(\mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, \overline{\mathsf{rep}_{\mathrm{A}}}(\ldots), \overline{\mathsf{rep}_{\mathrm{B}}}(\ldots), x))
\end{array}$$

The simultaneously defined $\overline{\mathsf{rep}_\mathrm{B}}$ is very simple:

$$\begin{aligned}
\overline{\mathsf{rep}_\mathrm{B}}(\gamma_\mathrm{A}, \mathsf{intro}_\mathrm{A}, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y}, \mathsf{a}_\mathrm{ref}(x), y) &= !(y) \\
\overline{\mathsf{rep}_\mathrm{B}}(\gamma_\mathrm{A}, \mathsf{intro}_\mathrm{A}, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y}, \mathsf{b}_\mathrm{ref}(x), \star) &= \mathsf{rep}_\mathrm{Y}(y) \\
\overline{\mathsf{rep}_\mathrm{B}}(\gamma_\mathrm{A}, \mathsf{intro}_\mathrm{A}, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y}, \mathsf{arg}(x), y) &= !(y)
\end{aligned}$$

Example 3.10 We define some terms in $\mathsf{A\text{-}Term}(\gamma_\rhd, X_\mathrm{ref}, Y_\mathrm{ref})$, where

$$\gamma_\rhd = \mathsf{A\text{-}ind}(\mathbf{1}, \mathsf{B\text{-}ind}(\mathbf{1}, \lambda(\star : \mathbf{1}).\,\mathsf{inr}(\star), \mathsf{nil}))$$

is the code for the constructor

$$\rhd : \big((\Sigma\Gamma : \mathbf{1} \to A)(\mathbf{1} \to B(\Gamma(\star))) \times \mathbf{1}\big) \to A \ .$$

Suppose that we have $\hat{a} : X_\mathrm{ref}$ with $\mathsf{rep}_\mathrm{X}(\hat{a}) = a : A$ and $\hat{b} : Y_\mathrm{ref}$ with $\mathsf{rep}_\mathrm{index}(\hat{b}) = a$ and $\mathsf{rep}_\mathrm{Y}(\hat{b}) = b : B(a)$. We then have

- $\mathsf{a}_\mathrm{ref}(\hat{a}) : \mathsf{A\text{-}Term}(\gamma_\rhd, X_\mathrm{ref}, Y_\mathrm{ref})$ with $\overline{\mathsf{rep}_\mathrm{A}}(\gamma_\rhd, \rhd, \ldots, \hat{a}) = a$ (so elements from X_ref are terms).
- $\mathsf{b}_\mathrm{ref}(\hat{b}) : \mathsf{A\text{-}Term}(\gamma_\rhd, X_\mathrm{ref}, Y_\mathrm{ref})$ with $\overline{\mathsf{rep}_\mathrm{A}}(\gamma_\rhd, \rhd, \ldots, \mathsf{b}_\mathrm{ref}(\hat{b})) = a$ (so elements from Y_ref are terms, representing the index of the element in B they represent). Furthermore $\overline{\mathsf{rep}_\mathrm{B}}(\gamma_\rhd, \rhd, \ldots, \mathsf{b}_\mathrm{ref}(\hat{b}), \star) = b$.
- $\overgroup{a \rhd b} := \mathsf{arg}(\langle(\lambda\,\star\,.\,\mathsf{b}_\mathrm{ref}(\hat{b})), \langle(\lambda\,\star\,.\,\star), \star\rangle\rangle) : \mathsf{A\text{-}Term}(\gamma_\rhd, X_\mathrm{ref}, Y_\mathrm{ref})$ with

$$\overline{\mathsf{rep}_\mathrm{A}}(\gamma_\rhd, \rhd, \ldots, \overgroup{a \rhd b}) = (\mathsf{rep}_\mathrm{index}(\hat{b})) \rhd (\mathsf{rep}_\mathrm{Y}(\hat{b})) = a \rhd b \ . \qquad \blacksquare$$

3.2.3.3 The universe SP^0_B of descriptions of B

We now introduce the universe SP_B of descriptions for B. It has formation rule

$$\frac{X_\mathrm{ref}, Y_\mathrm{ref} : \mathsf{Set} \qquad \gamma_\mathrm{A} : \mathsf{SP}^0_\mathrm{A}}{\mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})\ \mathsf{type}}$$

Again, we are interested in codes which initially do not refer to any elements and define $\gamma_\mathrm{A} : \mathsf{SP}^0_\mathrm{A} \vdash \mathsf{SP}^0_\mathrm{B}(\gamma_\mathrm{A})$ type by $\mathsf{SP}^0_\mathrm{B}(\gamma_\mathrm{A}) := \mathsf{SP}_\mathrm{B}(\mathbf{0}, \mathbf{0}, \gamma_\mathrm{A})$.

The introduction rules for SP_B are similar to the ones for SP_A. However, we now need to specify an index for the codomain of the constructor, and indices for arguments inductive in B can be arbitrary terms built up from $\mathsf{intro}_\mathrm{A}$ and elements we can refer to.

$$\frac{\overgroup{a} : \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref})}{\mathsf{nil}(\overgroup{a}) : \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})}$$

The code $\mathsf{nil}(\overgroup{a})$ represents a trivial constructor $c : \mathbf{1} \to B(a)$ (a base case), where the index a is encoded by $\overgroup{a} : \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref})$.

$$\frac{K : \mathsf{Set} \qquad \gamma : K \to \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})}{\mathsf{non\text{-}ind}(K, \gamma) : \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A}))}$$

The code $\mathsf{non\text{-}ind}(K,\gamma)$ represents a non-inductive argument $x : K$, with the rest of the arguments given by $\gamma(x)$.

$$\frac{K : \mathsf{Set} \qquad \gamma : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}} + K, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}})}{\mathsf{A\text{-}ind}(K,\gamma) : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}})}$$

The code $\mathsf{A\text{-}ind}(K,\gamma)$ represents an inductive argument with type $K \to A$, with the rest of the arguments given by γ.

$$\frac{K : \mathsf{Set} \qquad h_{\mathrm{index}} : K \to \mathsf{A\text{-}Term}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}) \qquad \gamma : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}} + K, \gamma_{\mathrm{A}})}{\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma) : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}})}$$

The code $\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma)$ represents an inductive argument of type $(x : K) \to B(i(x))$, where the index $i(x)$ is determined by h_{index}, and the rest of the arguments are given by γ. Notice how the index of the argument is now encoded by arbitrary terms in $\mathsf{A\text{-}Term}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}})$.

Example 3.11 The constructor

$$\Pi : \big((\Sigma\Gamma : \mathsf{Ctxt})(\Sigma\sigma : \mathsf{Ty}(\Gamma))\mathsf{Ty}(\Gamma \rhd \sigma)\big) \to \mathsf{Ty}(\Gamma)$$

is represented by the code

$$\gamma_{\Pi} = \mathsf{A\text{-}ind}(\mathbf{1}, \mathsf{B\text{-}ind}(\mathbf{1}, \lambda \star . \widehat{\Gamma}, \mathsf{B\text{-}ind}(\mathbf{1}, \lambda \star . \widehat{\mathsf{in}\langle\Gamma,\sigma\rangle}, \mathsf{nil}(\widehat{\Gamma}))))$$

where $\widehat{\Gamma} = \mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star))$ is the element representing the first argument $\Gamma : \mathsf{Ctxt}$ and $\widehat{\mathsf{in}\langle\Gamma,\sigma\rangle} = \mathsf{arg}(\langle(\lambda \star . \mathsf{b}_{\mathrm{ref}}(\mathsf{inr}(\star))), \langle \lambda \star . \star, \star\rangle\rangle)$ is the element representing $\Gamma \rhd \sigma$. ■

The definition of $\mathsf{Arg}_{\mathrm{B}}$ should now not come as a surprise. First, we have a formation rule:

$$\frac{\begin{matrix}\gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}} \\ X_{\mathrm{ref}}, Y_{\mathrm{ref}} : \mathsf{Set} \\ \gamma : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}})\end{matrix} \qquad \begin{matrix} X : \mathsf{Set} \\ Y : X \to \mathsf{Set} \\ \mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y) \to X\end{matrix} \qquad \begin{matrix}\mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X \\ \mathsf{rep}_{\mathrm{index}} : Y_{\mathrm{ref}} \to X \\ \mathsf{rep}_{\mathrm{Y}} : (x : Y_{\mathrm{ref}}) \to Y(\mathsf{rep}_{\mathrm{index}}(x))\end{matrix}}{\mathsf{Arg}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \gamma) : \mathsf{Set}}$$

The definition can be simplified for codes in $\mathsf{SP}^0_{\mathrm{B}}(\gamma_{\mathrm{A}})$:

$$\mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, \gamma) \coloneqq \mathsf{Arg}_{\mathrm{B}}(\mathbf{0}, \mathbf{0}, \gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, !_X, !_X, !_{Y \circ !}, \gamma)$$

We define[4]:

$$\begin{aligned}
&\mathsf{Arg}_{\mathrm{B}}(\_,\_,\_,\_,\_,\_,\_,\_,\_,\mathsf{nil}(\widehat{a})) = \mathbf{1}\\
&\mathsf{Arg}_{\mathrm{B}}(\_,\_,\_,\_,\_,\_,\_,\_,\_,\mathsf{non\text{-}ind}(K,\gamma)) = (\Sigma x : K)\mathsf{Arg}_{\mathrm{B}}(\_,\_,\_,\_,\_,\_,\_,\_,\_,\gamma(x))\\
&\mathsf{Arg}_{\mathrm{B}}(X_{\mathrm{ref}},\_,\_,X,\_,\_,\mathsf{rep}_{\mathrm{X}},\_,\_,\mathsf{A\text{-}ind}(K,\gamma))\\
&\quad = (\Sigma j : K \to X)\mathsf{Arg}_{\mathrm{B}}(X_{\mathrm{ref}} + K,\_,\_,\_,\_,\_,[\mathsf{rep}_{\mathrm{X}}, j],\_,\_,\gamma)\\
&\mathsf{Arg}_{\mathrm{B}}(\_,Y_{\mathrm{ref}},\gamma_{\mathrm{A}},\_,Y,\mathsf{intro}_{\mathrm{A}},\mathsf{rep}_{\mathrm{X}},\mathsf{rep}_{\mathrm{index}},\mathsf{rep}_{\mathrm{Y}},\mathsf{B\text{-}ind}(K,h_{\mathrm{index}},\gamma))\\
&\quad = (\Sigma j : (x : K) \to Y((\overline{\mathsf{rep}_{\mathrm{A}}}(\gamma_{\mathrm{A}},\mathsf{intro}_{\mathrm{A}},\mathsf{rep}_{\mathrm{X}},\mathsf{rep}_{\mathrm{index}},\mathsf{rep}_{\mathrm{Y}}) \circ h_{\mathrm{index}})(x)))\\
&\qquad\qquad \mathsf{Arg}_{\mathrm{B}}(\_,Y_{\mathrm{ref}} + K,\_,\_,\_,\_,\_,[\mathsf{rep}_{\mathrm{index}},\overline{\mathsf{rep}_{\mathrm{A}}}(\ldots) \circ h_{\mathrm{index}}],[\mathsf{rep}_{\mathrm{Y}}, j],\gamma)
\end{aligned}$$

[4] For readability, we have once again replaced arguments which are simply passed on with "_" in the recursive call, and likewise on the left hand side if the argument is not used otherwise.

Finally, we need the function $\mathsf{Index}^0_{\mathrm{B}}(\ldots) : \mathsf{Arg}_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, X, Y, \mathsf{intro}_{\mathrm{A}}) \to X$ which to each $b : \mathsf{Arg}_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, X, Y, \mathsf{intro}_{\mathrm{A}})$ assigns an index $a : X$ such that the element constructed from b is in $Y(a)$.

$$\frac{\begin{array}{ccc} \gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}} & & \mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X \\ X_{\mathrm{ref}}, Y_{\mathrm{ref}} : \mathsf{Set} & X : \mathsf{Set} \quad Y : X \to \mathsf{Set} & \mathsf{rep}_{\mathrm{index}} : Y_{\mathrm{ref}} \to X \\ \gamma : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}) & \mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y) \to X & \mathsf{rep}_{\mathrm{Y}} : (x : Y_{\mathrm{ref}}) \to Y(\mathsf{rep}_{\mathrm{index}}(x)) \end{array}}{\mathsf{Index}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \gamma) : \mathsf{Arg}_{\mathrm{B}}(\ldots) \to X}$$

For codes in $\mathsf{SP}^0_{\mathrm{B}}(\gamma_{\mathrm{A}})$, we define $\mathsf{Index}^0_{\mathrm{B}} : \mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, \gamma_{\mathrm{B}}) \to X$ by

$$\mathsf{Index}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, \gamma_{\mathrm{B}}) \coloneqq \mathsf{Index}_{\mathrm{B}}(\mathbf{0}, \mathbf{0}, \gamma_{\mathrm{A}}, X, Y, \mathsf{intro}_{\mathrm{A}}, !_X, !_X, !_{Y \circ !}, \gamma_{\mathrm{B}}) \ .$$

The equations by necessity follows the same pattern as the equations for $\mathsf{Arg}_{\mathrm{B}}$. For the base case $\gamma_{\mathrm{B}} = \mathsf{nil}(\widehat{a})$, we use $\mathsf{rep}_{\mathrm{X}}(\ldots, \widehat{a})$, and for the other cases, we just do a recursive call[5]

$$\begin{aligned} &\mathsf{Index}_{\mathrm{B}}(\_, \_, \gamma_{\mathrm{A}}, \_, \_, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{nil}(\widehat{a}), \star) \\ &\quad = \overline{\mathsf{rep}_{\mathrm{A}}}(\gamma_{\mathrm{A}}, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \widehat{a}) \\ &\mathsf{Index}_{\mathrm{B}}(\_, \_, \_, \_, \_, \_, \_, \_, \_, \mathsf{non\text{-}ind}(K, \gamma), \langle k, y \rangle) \\ &\quad = \mathsf{Index}_{\mathrm{B}}(\_, \_, \_, \_, \_, \_, \_, \_, \_, \gamma(k), y) \\ &\mathsf{Index}_{\mathrm{B}}(X_{\mathrm{ref}}, \_, \_, X, \_, \_, \mathsf{rep}_{\mathrm{X}}, \_, \_, \mathsf{A\text{-}ind}(K, \gamma), \langle j, y \rangle) \\ &\quad = \mathsf{Index}_{\mathrm{B}}(X_{\mathrm{ref}} + K, \_, \_, \_, \_, \_, [\mathsf{rep}_{\mathrm{X}}, j], \_, \_, \gamma, y) \\ &\mathsf{Index}_{\mathrm{B}}(\_, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}, \_, Y, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma), \langle j, y \rangle) \\ &\quad = \mathsf{Index}_{\mathrm{B}}(\_, Y_{\mathrm{ref}} + K, \_, \_, \_, \_, \_, [\mathsf{rep}_{\mathrm{index}}, \overline{\mathsf{rep}_{\mathrm{A}}}(\ldots) \circ h_{\mathrm{index}}], [\mathsf{rep}_{\mathrm{Y}}, j], \gamma, y) \end{aligned}$$

Example 3.12 Recall from Example 3.11 that the constructor $\Pi : \big((\Sigma\Gamma : \mathsf{Ctxt})(\Sigma\sigma : \mathsf{Ty}(\Gamma))\mathsf{Ty}(\Gamma \rhd \sigma)\big) \to \mathsf{Ty}(\Gamma)$ from Example 3.1 is represented by the code

$$\gamma_\Pi = \mathsf{A\text{-}ind}(\mathbf{1}, \mathsf{B\text{-}ind}(\mathbf{1}, (\lambda \star . \widehat{\Gamma}), \mathsf{B\text{-}ind}(\mathbf{1}, (\lambda \star . \widehat{\Gamma \rhd \sigma}, \mathsf{nil}(\widehat{\Gamma}))))) : \mathsf{SP}^0_{\mathrm{B}}(\gamma_\rhd) \ ,$$

where $\widehat{\Gamma} = \mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star)) : \mathsf{A\text{-}Term}(\mathbf{0} + \mathbf{1}, \mathbf{0}, \gamma_\rhd)$ and

$$\widehat{\Gamma \rhd \sigma} = \mathsf{arg}(\langle(\lambda \star . \mathsf{b}_{\mathrm{ref}}(\mathsf{inr}(\star))), \langle \lambda \star . \star, \star \rangle\rangle) : \mathsf{A\text{-}Term}(\mathbf{0} + \mathbf{1}, \mathbf{0} + \mathbf{1}, \gamma_\rhd) \ .$$

We have

$$\begin{aligned} &\mathsf{Arg}^0_{\mathrm{B}}(\gamma_\rhd, \mathsf{Ctxt}, \mathsf{Ty}, \rhd, \gamma_\Pi) = \\ &\qquad\qquad (\Sigma\Gamma : \mathbf{1} \to \mathsf{Ctxt})(\Sigma\sigma : \mathbf{1} \to \mathsf{Ty}(\Gamma(\star)))(\mathbf{1} \to \mathsf{Ty}(\Gamma(\star) \rhd \sigma(\star))) \times \mathbf{1} \end{aligned}$$

and $\mathsf{Index}^0_{\mathrm{B}}(\gamma_\rhd, \mathsf{Ctxt}, \mathsf{Ty}, \rhd, \gamma_\Pi, \langle \Gamma, \sigma, \tau, \star \rangle) = \Gamma(\star)$. ■

[5] Simply passed on and otherwise not used arguments have been replaced with "_" for readability.

3.2.3.4 Formation and introduction rules

We are now ready to give the formation and introduction rules for A and B. They all have the common premises $\gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}}$, $\gamma_{\mathrm{B}} : \mathsf{SP}^0_{\mathrm{B}}(\gamma_{\mathrm{A}})$, which will be omitted.

Axioms 3.13 (Formation and introduction rules for inductive-inductive definitions) Formation rules:

$$A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : \mathsf{Set} \qquad B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} \to \mathsf{Set}$$

Introduction rule for $A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{a : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}})}{\mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(a) : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}$$

Introduction rule for $B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{b : \mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}, \gamma_{\mathrm{B}})}{\mathsf{intro}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(b) : B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}(\mathsf{Index}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}, \gamma_{\mathrm{B}}, b))} \quad ■$$

3.2.3.5 Derived rules for convenience

Encoding multiple constructors into one The theory we have presented assumes that both A and B have exactly one constructor each. This is no limitation, as multiple constructors can always be encoded into one by using non-inductive arguments. Suppose that $\mathsf{intro}_0 : F_0(A, B) \to A$ and $\mathsf{intro}_1 : F_1(A, B) \to A$ are two constructors for A. Then we can combine them into one constructor

$$\mathsf{intro}_{0+1} : \big((\Sigma i : \mathbf{2})F_i(A, B)\big) \to A$$

by defining $\mathsf{intro}_{0+1}(i, x) = \mathsf{intro}_i(x)$. Of course, this is only possible because we have dependent types at our disposal.

If intro_0 is described by the code γ_0 and intro_1 by γ_1, then intro_{0+1} is described by the code

$$\gamma_0 +_{\mathsf{SP}} \gamma_1 := \mathsf{non\text{-}ind}(\mathbf{2}, \lambda x.\,\mathsf{if}\ x\ \mathsf{then}\ \gamma_0\ \mathsf{else}\ \gamma_1)\ .$$

Notice that this makes use of large elimination for Booleans $\mathbf{2}$ in an essential way, as $\mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})$ is a large type.

Single inductive arguments An inductive arguments is always of the form $K \to A$ or $(x : K) \to B(i(x))$ for some set K of premises (and index function $i : K \to A$). If we only want a single inductive argument, we choose $K = \mathbf{1}$. For convenience, we can define

$$\begin{aligned}
&\mathsf{A\text{-}ind}_1 : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + \mathbf{1}) \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})\\
&\mathsf{B\text{-}ind}_1 : X_{\mathrm{ref}} \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}}) \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})
\end{aligned}$$

and the SP_B variants

$$\begin{aligned}
&\mathsf{A\text{-}ind}_\mathbf{1} : \mathsf{SP}_\mathrm{B}(X_\mathrm{ref} + \mathbf{1}, Y_\mathrm{ref}, \gamma_\mathrm{A}) \to \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})\\
&\mathsf{B\text{-}ind}_\mathbf{1} : \mathsf{A\text{-}Term}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A}) \to \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref} + \mathbf{1}, \gamma_\mathrm{A}) \to \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})
\end{aligned}$$

by

$$\begin{aligned}
\mathsf{A\text{-}ind}_\mathbf{1}(\gamma) &= \mathsf{A\text{-}ind}(\mathbf{1}, \gamma)\\
\mathsf{B\text{-}ind}_\mathbf{1}(i, \gamma) &= \mathsf{B\text{-}ind}(\mathbf{1}, \lambda\_.\, i, \gamma)
\end{aligned}$$

This is possible since we have η-rules for $\mathbf{1}$, so that the index $i : \mathsf{A\text{-}Term}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})$ cannot depend on $x : \mathbf{1}$ in any way different from $\star : \mathbf{1}$. By using this abbreviation, we can make the codes slightly more readable.

Non-dependent non-inductive arguments Later arguments may depend on non-inductive arguments. In case they do not, we introduce the abbreviations

$$\begin{aligned}
&\mathsf{non\text{-}ind}' : (K : \mathsf{Set}) \to \mathsf{SP}_\mathrm{A}(X_\mathrm{ref}) \to \mathsf{SP}_\mathrm{A}(X_\mathrm{ref})\\
&\mathsf{non\text{-}ind}' : (K : \mathsf{Set}) \to \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A}) \to \mathsf{SP}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A})
\end{aligned}$$

for

$$\mathsf{non\text{-}ind}'(K, \gamma) = \mathsf{non\text{-}ind}(K, \lambda\_.\, \gamma) \ .$$

3.2.4 The examples revisited

We show how to find γ_A, γ_B for some well-known sets, including the examples in Section 3.1.

3.2.4.1 Well-orderings

Ordinary inductive definitions can be interpreted as inductive-inductive definitions where we only care about the index set A and not about the family $B : A \to \mathsf{Set}$. A canonical choice is to let B have constructor $\mathsf{intro}_\mathrm{B} : (x : A) \to B(x)$, which is described by the code $\gamma_\mathrm{dummy} := \mathsf{A\text{-}ind}(\mathbf{1}, \mathsf{nil}(\mathsf{a}_\mathrm{ref}(\mathsf{inr}(\star))))$[6].

For every $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$, let

$$\gamma_{W(A,B)} := \mathsf{non\text{-}ind}(A, \lambda x.\, \mathsf{A\text{-}ind}(B(x), \mathsf{nil}))$$

and define $W(A,B) := A_{\gamma_{W(A,B)}, \gamma_\mathrm{dummy}}$. Then $W(A,B)$ has constructor

$$\mathsf{intro}_{W(A,B)} : \big((\Sigma x : A)(B(x) \to W(A,B)) \times \mathbf{1}\big) \to W(A,B) \ .$$

[6] Another choice is $\gamma_\mathrm{dummy} = \mathsf{non\text{-}ind}(\mathbf{0}, !_{\mathsf{SP}^0_\mathrm{B}(\gamma_\mathrm{A})})$, which makes $B(x)$ an empty type.

3.2.4.2 Finite sets

Also indexed inductive definitions can be interpreted as inductive-inductive definitions. We simply let the index set A be an isomorphic copy of the fixed index set I from the indexed inductive definition (i.e. A is given by the constructor $\mathsf{intro}_{\mathrm{A}} : I \to A$).

For the family $\mathsf{Fin} : \mathbb{N} \to \mathsf{Set}$ of finite sets, the index set is $\mathbb{N}$, so we define

$$\gamma_A \coloneqq \mathsf{non\text{-}ind}'(\mathbb{N}, \mathsf{nil}) : \mathsf{SP}^0_{\mathrm{A}}$$

and

$$\gamma_{\mathsf{Fin}} \coloneqq \gamma_{\mathsf{z}} +_{\mathsf{SP}} \gamma_{\mathsf{s}} : \mathsf{SP}^0_{\mathrm{B}}(\gamma_{\mathrm{A}})$$

where

$$\begin{aligned}
\gamma_{\mathsf{z}} &\coloneqq \mathsf{non\text{-}ind}(\mathbb{N}, \lambda n.\, \mathsf{nil}(\mathsf{arg}(\langle n+1, \star\rangle))) \ , \\
\gamma_{\mathsf{s}} &\coloneqq \mathsf{non\text{-}ind}(\mathbb{N}, \lambda n.\, \mathsf{B\text{-}ind}_{\mathbf{1}}(\mathsf{arg}(\langle n, \star\rangle), \mathsf{nil}(\mathsf{arg}(\langle n+1, \star\rangle)))) \ .
\end{aligned}$$

Then the constructor $\mathsf{intro}_{A_{\gamma_A,\gamma_{\mathsf{Fin}}}} : (\mathbb{N} \times \mathbf{1}) \to A_{\gamma_A,\gamma_{\mathsf{Fin}}}$ is one part of an isomorphism $\mathbb{N} \cong \mathbb{N} \times \mathbf{1} \cong A_{\gamma_A,\gamma_{\mathsf{Fin}}}$, and if we define $\mathsf{Fin} : \mathbb{N} \to \mathsf{Set}$ by

$$\mathsf{Fin}(n) = B_{\gamma_A,\gamma_{\mathsf{Fin}}}(\mathsf{intro}_{A_{\gamma_A,\gamma_{\mathsf{Fin}}}}(\langle n, \star\rangle)) \ ,$$

then we can define constructors

$$\frac{n : \mathbb{N}}{\mathsf{z}_n : \mathsf{Fin}(n+1)} \qquad \frac{n : \mathbb{N} \qquad m : \mathsf{Fin}(n)}{\mathsf{s}_n(m) : \mathsf{Fin}(n+1)}$$

by $\mathsf{z}_n = \mathsf{intro}_{B_{\gamma_A,\gamma_{\mathsf{Fin}}}}(\langle \mathsf{tt}, \langle n, \star\rangle\rangle)$ and

$$\mathsf{s}_n(m) = \mathsf{intro}_{B_{\gamma_A,\gamma_{\mathsf{Fin}}}}(\langle \mathsf{ff}, \langle n, \langle(\lambda \star . m), \star\rangle\rangle\rangle \ .$$

3.2.4.3 Contexts and types

The codes for the contexts and types from Example 3.1 are as follows:

$$\begin{aligned}
\gamma_{\mathsf{Ctxt}} &= \mathsf{nil} +_{\mathsf{SP}} \mathsf{A\text{-}ind}_{\mathbf{1}}(\mathsf{B\text{-}ind}_{\mathbf{1}}(\mathsf{inr}(\star), \mathsf{nil})) : \mathsf{SP}^0_{\mathrm{A}} \\
\gamma_{\iota} &= \mathsf{A\text{-}ind}_{\mathbf{1}}(\mathsf{nil}(\mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star)))) \\
\gamma_{\Pi} &= \mathsf{A\text{-}ind}_{\mathbf{1}}(\mathsf{B\text{-}ind}_{\mathbf{1}}(\mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star)), \mathsf{B\text{-}ind}_{\mathbf{1}}(\mathsf{arg}(\langle \mathsf{ff}, \langle(\lambda \star . \mathsf{b}_{\mathrm{ref}}(\mathsf{inr}(\star))), \langle \lambda \star . \star, \star\rangle\rangle\rangle), \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{nil}(\mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star)))))) \\
\gamma_{\mathsf{Ty}} &= \gamma_{\iota} +_{\mathsf{SP}} \gamma_{\Pi} : \mathsf{SP}^0_{\mathrm{B}}(\gamma_{\mathsf{Ctxt}}) \ .
\end{aligned}$$

We have $\mathsf{Ctxt} = A_{\gamma_{\mathsf{Ctxt}},\gamma_{\mathsf{Ty}}}$ and $\mathsf{Ty} = B_{\gamma_{\mathsf{Ctxt}},\gamma_{\mathsf{Ty}}}$ and we can define the usual constructors by

$$\begin{aligned}
&\varepsilon : \mathsf{Ctxt} && \iota : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \\
&\varepsilon = \mathsf{intro}_{A_{\gamma_{\mathsf{Ctxt}},\gamma_{\mathsf{Ty}}}}(\langle \mathsf{tt}, \star\rangle) \ , && \iota_\Gamma = \mathsf{intro}_{B_{\gamma_{\mathsf{Ctxt}},\gamma_{\mathsf{Ty}}}}(\langle \mathsf{tt}, \langle(\lambda \star . \Gamma), \star\rangle\rangle) \ ,
\end{aligned}$$

$$\begin{aligned}
&\rhd : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Ctxt} \\
&\Gamma \rhd \sigma = \mathsf{intro}_{A_{\gamma_{\mathsf{Ctxt}},\gamma_{\mathsf{Ty}}}}(\langle \mathsf{ff}, \langle(\lambda \star . \Gamma), \langle(\lambda \star . \sigma), \star\rangle\rangle\rangle) \ ,
\end{aligned}$$

$$\begin{aligned}
&\Pi : (\Gamma : \mathsf{Ctxt}) \to (\sigma : \mathsf{Ty}(\Gamma)) \to \mathsf{Ty}(\Gamma \rhd \sigma) \to \mathsf{Ty}(\Gamma) \\
&\Pi(\Gamma, \sigma, \tau) = \mathsf{intro}_{B_{\gamma_{\mathsf{Ctxt}},\gamma_{\mathsf{Ty}}}}(\langle \mathsf{ff}, \langle(\lambda \star . \Gamma), \langle(\lambda \star . \sigma), \langle(\lambda \star . \tau), \star\rangle\rangle\rangle\rangle) \ .
\end{aligned}$$

3.2.5 Elimination rules

The fact that the sets just introduced are inductive is encoded in the elimination rules. Intuitively, they state that a function from an inductive-inductive definition $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$ is determined by its values on constructors. Since functions can be dependent, this also gives a way to do proofs by induction for inductive-inductive definitions. How dependent should these functions be? The first thought that comes to mind is to consider motives of the form

$$\begin{aligned} &P : A \to \mathsf{Set} \\ &Q : (x : A) \to B(x) \to \mathsf{Set} \end{aligned}$$

so that the elimination rules give rise to functions

$$\begin{aligned} &\mathsf{elim}'_A : \ldots \to (x : A) \to P(x) \\ &\mathsf{elim}'_B : \ldots \to (x : A) \to (y : B(x)) \to Q(x, y) \end{aligned}$$

We call these elimination rules *simple* (not to be confused with simply typed, or non-dependent elimination rules!), and formalise them in Section 3.2.5.2. Sometimes, however, we need a more general notion of elimination rules where the motive has the form

$$\begin{aligned} &P : A \to \mathsf{Set} \\ &Q : (x : A) \to B(x) \to P(x) \to \mathsf{Set} \end{aligned}$$

(notice the dependency of Q on P). In this case, the elimination rules gives rise to functions

$$\begin{aligned} &\mathsf{elim}_A : \ldots \to (x : A) \to P(x) \\ &\mathsf{elim}_B : \ldots \to (x : A) \to (y : B(x)) \to Q(x, y, \mathsf{elim}_A(\ldots, x)) \end{aligned}$$

where elim_A appears in the type of elim_B. We could summarise the situation in the following slogan:

The elimination principle for inductive-inductive definitions is recursive-recursive.

We call these elimination rules the *general* rules. A closed axiomatisation of these rules seem to require at least function extensionality (on the other hand, for closed inductive-inductive definitions defined in the empty context, no such further assumptions are necessary). Instead of giving these more involved rules, we will return to the general elimination rules in Chapter 4, but from a high-level, categorical point of view.

3.2.5.1 Examples of elimination rules

Consider the data type of sorted lists[7] from Example 3.2.

[7] The inductive-inductive definition of the data type of sorted lists falls outside the axiomatisation presented in this chapter, as remarked at the end of Section 3.1. This will be justified in Section 6.2.1.

Example 3.14 (Simple elimination rules for sorted lists)

$$\begin{aligned}
\mathsf{elim}'_{\mathsf{SortedList}} : {} & (P : \mathsf{SortedList} \to \mathsf{Set}) \to \\
& (Q : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to n \leq_{\mathrm{L}} \ell \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\mathsf{nil}} : P(\mathsf{nil})) \to \\
& \big(\mathsf{step}_{\mathsf{cons}} : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \to P(\ell) \\
& \qquad \to Q(n, \ell, p) \to P(\mathsf{cons}(n, \ell, p))\big) \to \\
& \big(\mathsf{step}_{\mathsf{triv}} : (m : \mathbb{N}) \to Q(m, \mathsf{nil}, \mathsf{triv}_n)\big) \to \\
& \big(\mathsf{step}_{\ll \cdot \gg} : (m : \mathbb{N}) \to (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \\
& \qquad \to (q : m \leq n) \to (p' : m \leq_{\mathrm{L}} \ell) \to P(\ell) \\
& \qquad \to Q(n, \ell, p) \to Q(m, \ell, p') \to Q(m, \mathsf{cons}(n, \ell, p), \ll q, p' \gg)\big) \to \\
& (\ell : \mathsf{SortedList}) \to P(\ell) \enspace , \\
\mathsf{elim}'_{\leq_{\mathrm{L}}} : {} & (P : \mathsf{SortedList} \to \mathsf{Set}) \to \\
& (Q : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to n \leq_{\mathrm{L}} \ell \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\mathsf{nil}} : \ldots) \to \\
& \big(\mathsf{step}_{\mathsf{cons}} : \ldots\big) \to \\
& \big(\mathsf{step}_{\mathsf{triv}} : \ldots\big) \to \\
& \big(\mathsf{step}_{\ll \cdot \gg} : \ldots\big) \to \\
& (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \to Q(n, \ell, p)
\end{aligned}$$

with computation rules

$$\mathsf{elim}'_{\mathsf{SortedList}}(P, Q, \mathsf{step}_{\mathsf{nil}}, \mathsf{step}_{\mathsf{cons}}, \mathsf{step}_{\mathsf{triv}}, \mathsf{step}_{\ll \cdot \gg}, \mathsf{nil}) = \mathsf{step}_{\mathsf{nil}} : P(\mathsf{nil})$$

and

$$\begin{aligned}
& \mathsf{elim}'_{\mathsf{SortedList}}(P, Q, \mathsf{step}_{\mathsf{nil}}, \mathsf{step}_{\mathsf{cons}}, \mathsf{step}_{\mathsf{triv}}, \mathsf{step}_{\ll \cdot \gg}, \mathsf{cons}(n, \ell, p)) \\
& \qquad = \mathsf{step}_{\mathsf{cons}}(n, \ell, p, \mathsf{elim}_{\mathsf{SortedList}}(\ldots, \ell), \mathsf{elim}_{\leq_{\mathrm{L}}}(\ldots, n, \ell, p)) : P(\mathsf{cons}(n, \ell, p))
\end{aligned}$$

for $\mathsf{elim}'_{\mathsf{SortedList}}$, and

$$\mathsf{elim}'_{\leq_{\mathrm{L}}}(P, Q, \mathsf{step}_{\mathsf{nil}}, \mathsf{step}_{\mathsf{cons}}, \mathsf{step}_{\mathsf{triv}}, \mathsf{step}_{\ll \cdot \gg}, m, \mathsf{nil}, \mathsf{triv}_m) = \mathsf{step}_{\mathsf{triv}}(m) : Q(m, \mathsf{nil}, \mathsf{triv}_m)$$

and

$$\begin{aligned}
& \mathsf{elim}'_{\leq_{\mathrm{L}}}(P, Q, \mathsf{step}_{\mathsf{nil}}, \mathsf{step}_{\mathsf{cons}}, \mathsf{step}_{\mathsf{triv}}, \mathsf{step}_{\ll \cdot \gg}, m, \mathsf{cons}(n, \ell, p), \ll q, p' \gg) \\
& \qquad = \mathsf{step}_{\ll \cdot \gg}(m, n, \ell, p, q, p', \mathsf{elim}_{\mathsf{SortedList}}(\ldots, \ell), \\
& \qquad\qquad \mathsf{elim}_{\leq_{\mathrm{L}}}(\ldots, n, \ell, p), \mathsf{elim}_{\leq_{\mathrm{L}}}(\ldots, m, \ell, p'))
\end{aligned}$$

for $\mathsf{elim}_{\leq_{\mathrm{L}}}$. ■

Example 3.15 (General elimination rules for sorted lists)

$$\begin{aligned}
\mathsf{elim}_{\mathsf{SortedList}} : {} & (P : \mathsf{SortedList} \to \mathsf{Set}) \to \\
& (Q : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to n \leq_{\mathrm{L}} \ell \to P(\ell) \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\mathsf{nil}} : P(\mathsf{nil})) \to \\
& \big(\mathsf{step}_{\mathsf{cons}} : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \to (\widetilde{\ell} : P(\ell)) \\
& \qquad \to Q(n, \ell, p, \widetilde{\ell}) \to P(\mathsf{cons}(n, \ell, p))\big) \to \\
& \big(\mathsf{step}_{\mathsf{triv}} : (m : \mathbb{N}) \to Q(m, \mathsf{nil}, \mathsf{triv}_n, \mathsf{step}_{\mathsf{nil}})\big) \to \\
& \big(\mathsf{step}_{\ll \cdot \gg} : (m : \mathbb{N}) \to (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \\
& \qquad \to (q : m \leq n) \to (p' : m \leq_{\mathrm{L}} \ell) \to (\widetilde{\ell} : P(\ell)) \\
& \qquad \to (\widetilde{p} : Q(n, \ell, p, \widetilde{\ell})) \to (\widetilde{p'} : Q(m, \ell, p', \widetilde{\ell})) \\
& \qquad \to Q(m, \mathsf{cons}(n, \ell, p), \ll q, p' \gg, \mathsf{step}_{\mathsf{cons}}(n, \ell, p, \widetilde{\ell}, \widetilde{p}))\big) \to \\
& (\ell : \mathsf{SortedList}) \to P(\ell) \ ,
\end{aligned}$$

$$\begin{aligned}
\mathsf{elim}_{\leq_{\mathrm{L}}} : {} & (P : \mathsf{SortedList} \to \mathsf{Set}) \to \\
& (Q : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to n \leq_{\mathrm{L}} \ell \to P(\ell) \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\mathsf{nil}} : \ldots) \to \\
& \big(\mathsf{step}_{\mathsf{cons}} : \ldots\big) \to \\
& \big(\mathsf{step}_{\mathsf{triv}} : \ldots\big) \to \\
& \big(\mathsf{step}_{\ll \cdot \gg} : \ldots\big) \to \\
& (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \\
& \qquad \to Q(n, \ell, p, \mathsf{elim}_{\mathsf{SortedList}}(\ldots, \ell)) \ .
\end{aligned}$$

The computation rules are the same for both the simple and general elimination rules, except that computation rules for the general eliminator $\mathsf{elim}_{\leq_{\mathrm{L}}}$ are well-typed only because of the computation rules for $\mathsf{elim}_{\mathsf{SortedList}}$. ■

Suppose that we want to define a function $\mathsf{insert} : \mathsf{SortedList} \to \mathbb{N} \to \mathsf{SortedList}$ which inserts a number m into its appropriate place in a sorted list ℓ to create a new sorted list $\mathsf{insert}(\ell, m)$. From a high-level perspective, this is easy: the elimination rules allows us to make case distinctions between empty and non-empty lists, so it suffices to handle these two cases separately. The empty list is easy to handle, and for non-empty lists, we compare m with the first element n of the list $\ell = [n, \ldots]$, which is possible since $\leq$ on natural numbers is decidable. If $m \leq n$, the result should be $[m, n, \ldots]$, otherwise we recursively insert m into the tail of the list.

In detail, we choose the motive $P(\ell) := \mathbb{N} \to \mathsf{SortedList}$ and, in our first attempt, we also choose the motive $Q(n, \ell, p, \widetilde{\ell}) := \mathbf{1}$, since we are only interested in getting a function $\mathsf{elim}_{\mathsf{SortedList}}(\ldots) : \mathsf{SortedList} \to \mathbb{N} \to \mathsf{SortedList}$. We need to give functions

$\mathsf{step}_{\mathsf{nil}} : (m : \mathbb{N}) \to \mathsf{SortedList}$ and $\mathsf{step}_{\mathsf{cons}}(n, \ell, p) : (\widetilde{\ell} : \mathbb{N} \to \mathsf{SortedList}) \to Q(n, \ell, p, \widetilde{\ell}) \to (m : \mathbb{N}) \to \mathsf{SortedList}$ to use when inserting into the empty list or the list $\mathsf{cons}(n, \ell, p)$ respectively. The argument $\widetilde{\ell} : \mathbb{N} \to \mathsf{SortedList}$ gives the result of a recursive call on ℓ.

The function $\mathsf{step}_{\mathsf{nil}}$ is easy to define: it should be

$$\mathsf{step}_{\mathsf{nil}}(m) = \mathsf{cons}(m, \mathsf{nil}, \mathsf{triv}_m)$$

For $\mathsf{step}_{\mathsf{cons}}$, the decidability of $\leq$ (combined with the fact that $\leq$ is total) allows us to distinguish between the cases when $m \leq n$ and $n \leq m$, and we are entitled to a proof $q : m \leq n$ or $q : n \leq m$ of this fact. We try:

$$\begin{aligned}&\mathsf{step}_{\mathsf{cons}}(n, \ell, p, \widetilde{\ell}, \star, m)\\ &\qquad = \begin{cases}\mathsf{cons}(m, \mathsf{cons}(n, \ell, p), \ll q, \mathsf{trans}_{\leq_{\mathrm{L}}}(q, p) \gg) & \text{where } q : m \leq n\\ \mathsf{cons}(n, \widetilde{\ell}(m), \boxed{\{?\}}) & \text{where } q : n \leq m\end{cases}\end{aligned}$$

Here, $\mathsf{trans}_{\leq_{\mathrm{L}}} : m \leq n \to n \leq_{\mathrm{L}} \ell \to m \leq_{\mathrm{L}} \ell$ witnesses a kind of transitivity of $\leq$ and $\leq_{\mathrm{L}}$. It can be straightforwardly defined with the elimination rules. The question is what we should fill the hole $\boxed{\{?\}}$ with. We need to provide a proof that $n \leq_{\mathrm{L}} \widetilde{l}(m)$, i.e. that $n \leq_{\mathrm{L}} \mathsf{insert}(l, m)$, if we remember that $\widetilde{l}$ is the result of the recursive call on ℓ. We need to prove this simultaneously as we define insert! Fortunately, this is exactly what the general elimination rules allow us to do if we choose a more meaningful Q.

Thus, we try again, but this time with

$$Q(n, \ell, p, \widetilde{\ell}) \coloneqq (m : \mathbb{N}) \to n \leq m \to n \leq_{\mathrm{L}} \widetilde{l}(m) \ .$$

Note that this would not have been possible with the simple elimination rules. The argument $\star : \mathbf{1}$ to $\mathsf{step}_{\mathsf{cons}}$ in our first attempt has now been replaced with the argument $\widetilde{p} : (m : \mathbb{N}) \to n \leq m \to n \leq_{\mathrm{L}} \widetilde{l}(m)$, and we can define

$$\mathsf{step}_{\mathsf{cons}}(n, \ell, p, \widetilde{\ell}, \widetilde{p}, m) = \begin{cases}\mathsf{cons}(m, \mathsf{cons}(n, \ell, p), \ll q, \mathsf{trans}_{\leq_{\mathrm{L}}}(q, p) \gg) & \text{where } q : m \leq n\\ \mathsf{cons}(n, \widetilde{\ell}(m), \widetilde{p}(m, q)) & \text{where } q : n \leq m\end{cases}$$

Now we must also define $\mathsf{step}_{\mathsf{triv}} : (n : \mathbb{N}) \to Q(n, \mathsf{nil}, \mathsf{triv}_n, \mathsf{step}_{\mathsf{nil}})$ and $\mathsf{step}_{\ll \cdot \gg}$ with type as above for our choice of P and Q. This presents us with no further difficulties. For $\mathsf{step}_{\mathsf{triv}}$, expanding $Q(n, \mathsf{nil}, \mathsf{triv}_n, \mathsf{step}_{\mathsf{nil}})$ and replacing $\mathsf{step}_{\mathsf{nil}}$ with its definition, we see that we should give a function of type

$$\mathsf{step}_{\mathsf{triv}} : (n : \mathbb{N}) \to (m : \mathbb{N}) \to n \leq m \to n \leq_{\mathrm{L}} \mathsf{cons}(m, \mathsf{nil}, \mathsf{triv}_m) \ ,$$

so we can define $\mathsf{step}_{\mathsf{triv}}(n, m, p) = \ll p, \mathsf{triv}_n \gg$. The definition of $\mathsf{step}_{\ll \cdot \gg}$ follows the pattern of $\mathsf{step}_{\mathsf{cons}}$ above. Rather than trying to explain it, we just give the definition:

$$\mathsf{step}_{\ll \cdot \gg}(m, n, \ell, p, q, p', \widetilde{\ell}, \widetilde{p}, \widetilde{p'}, x, r) = \begin{cases}\ll r, \ll q, p' \gg\gg & \text{where } s : m \leq n\\ \ll q, \widetilde{p'}(x, r) \gg & \text{where } s : n \leq m\end{cases}$$

With all pieces in place, we can now define $\mathsf{insert} : \mathsf{SortedList} \to \mathbb{N} \to \mathsf{SortedList}$ as $\mathsf{insert} = \mathsf{elim}_{\mathsf{SortedList}}(P, Q, \mathsf{step}_{\mathsf{nil}}, \mathsf{step}_{\mathsf{cons}}, \mathsf{step}_{\mathsf{triv}}, \mathsf{step}_{\ll \cdot \gg})$.

Table 3.2: Concepts involved in the elimination rules.

Name	Meaning
IH_{A}, IH_{B}	sets of inductive hypotheses
$\mathsf{map}_{\mathsf{IH}_{\mathrm{A}}}$, $\mathsf{map}_{\mathsf{IH}_{\mathrm{B}}}$	recursive calls
$\mathsf{elim}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}$, $\mathsf{elim}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}$	eliminators

3.2.5.2 Simple elimination rules

We now present the axiomatisation of the simple elimination rules, which follows the presentation in Section 3.2.2 closely. The concepts involved are summarised in Table 3.2.

We first define

$$\dfrac{\begin{array}{c} X_{\mathrm{ref}} : \mathsf{Set} \\ \gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}}) \end{array} \quad \begin{array}{c} X : \mathsf{Set} \\ Y : X \to \mathsf{Set} \\ \mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X \end{array} \quad \begin{array}{c} P : X \to \mathsf{Set} \\ Q : (x : X) \to Y(x) \to \mathsf{Set} \end{array} \quad x : \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}})}{\mathsf{IH}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}}, P, Q, x) : \mathsf{Set}}$$

and

$$\dfrac{\ldots \quad \begin{array}{c} P : X \to \mathsf{Set} \\ Q : (x : X) \to Y(x) \to \mathsf{Set} \end{array} \quad x : \mathsf{Arg}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}, X, Y, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \gamma_{\mathrm{B}})}{\mathsf{IH}_{\mathrm{B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma, X, Y, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, P, Q, x) : \mathsf{Set}}$$

by induction over γ and γ_{B} respectively[8]:

$$\begin{aligned}
\mathsf{IH}_{\mathrm{A}}(\ldots, \mathsf{nil}, P, Q, \star) &= \mathbf{1} \\
\mathsf{IH}_{\mathrm{A}}(\ldots, \mathsf{non\text{-}ind}(K, \gamma), P, Q, \langle k, x \rangle) &= \mathsf{IH}_{\mathrm{A}}(\gamma(k), P, Q, x) \\
\mathsf{IH}_{\mathrm{A}}(\ldots, \mathsf{A\text{-}ind}(K, \gamma), \ldots, P, Q, \langle j, x \rangle) &= ((k : K) \to P(j(k))) \times \mathsf{IH}_{\mathrm{A}}(\ldots, \gamma, \ldots, P, Q, x) \\
\mathsf{IH}_{\mathrm{A}}(\ldots, \mathsf{B\text{-}ind}(K, h, \gamma), \ldots, P, Q, \langle j, x \rangle) &= \\
&\quad ((k : K) \to Q(\mathsf{rep}_{\mathrm{X}}(h(k)), j(k))) \times \mathsf{IH}_{\mathrm{A}}(\ldots, \gamma, \ldots, P, Q, x)
\end{aligned}$$

$$\begin{aligned}
\mathsf{IH}_{\mathrm{B}}(\ldots, \mathsf{nil}(a), P, Q, \star) &= \mathbf{1} \\
\mathsf{IH}_{\mathrm{B}}(\ldots, \mathsf{non\text{-}ind}(K, \gamma), P, Q, \langle k, x \rangle) &= \mathsf{IH}_{\mathrm{B}}(\gamma(k), P, Q, x) \\
\mathsf{IH}_{\mathrm{B}}(\ldots, \mathsf{A\text{-}ind}(K, \gamma), \ldots, P, Q, \langle j, x \rangle) &= ((k : K) \to P(j(k))) \times \mathsf{IH}_{\mathrm{B}}(\ldots, \gamma, \ldots, P, Q, x) \\
\mathsf{IH}_{\mathrm{B}}(\ldots, \mathsf{B\text{-}ind}(K, h, \gamma), \ldots, P, Q, \langle j, x \rangle) &= \\
&\quad ((k : K) \to Q(\overline{\mathsf{rep}_{\mathrm{A}}}(\ldots, h(k)), j(k))) \times \mathsf{IH}_{\mathrm{B}}(\ldots, \gamma, \ldots, P, Q, x)
\end{aligned}$$

Note that these two sets are completely independent of one another. We now define functions $\mathsf{map}_{\mathsf{IH}_{\mathrm{A}}}$ and $\mathsf{map}_{\mathsf{IH}_{\mathrm{B}}}$ which take care of the recursive calls. The first function $\mathsf{map}_{\mathsf{IH}_{\mathrm{A}}}$ has the following type:

$$\dfrac{\ldots \quad \begin{array}{c} f : (x : X) \to P(x) \\ g : (x : X) \to (y : Y(x)) \to Q(x, y) \end{array} \quad x : \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}})}{\mathsf{map}_{\mathsf{IH}_{\mathrm{A}}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}}, P, Q, f, g, x) : \mathsf{IH}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}}, P, Q, x)}$$

[8] We have suppressed arguments that are handled in the same way as for $\mathsf{Arg}_{\mathrm{A}}$ and $\mathsf{Arg}_{\mathrm{B}}$ respectively.

while $\mathsf{map}_{\mathsf{IH}_\mathrm{B}}$ is of type

$$\frac{\begin{array}{c} f : (x : X) \to P(x) \\ \ldots \quad g : (x : X) \to (y : Y(x)) \to Q(x, y) \quad x : \mathsf{Arg}_\mathrm{B}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A}, X, Y, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y}, \gamma_\mathrm{B}) \end{array}}{\mathsf{map}_{\mathsf{IH}_\mathrm{B}}(X_\mathrm{ref}, Y_\mathrm{ref}, \gamma_\mathrm{A}, X, Y, \mathsf{intro}_\mathrm{A}, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y}, P, Q, \gamma_\mathrm{B}, f, g, x) : \mathsf{IH}_\mathrm{B}(X_\mathrm{ref}, \gamma, X, Y, \mathsf{rep}_\mathrm{X}, P, Q, x)}$$

The defining equations are:

$$\begin{aligned}
\mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\ldots, \mathsf{nil}, f, g, \star) &= \star \\
\mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\ldots, \mathsf{non\text{-}ind}(K, \gamma), f, g, \langle k, x\rangle) &= \mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\gamma(k), f, g, x) \\
\mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\ldots, \mathsf{A\text{-}ind}(K, \gamma), \ldots, f, g, \langle j, x\rangle) &= \langle f \circ j, \mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\ldots, \gamma, \ldots, f, g, x)\rangle \\
\mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\ldots, \mathsf{B\text{-}ind}(K, h, \gamma), \ldots, f, g, \langle j, x\rangle) &= \\
&\langle \lambda k.\, g(\mathsf{rep}_\mathrm{X}(h(k)), j(k)), \mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\ldots, \gamma, \ldots, f, g, x)\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\ldots, \mathsf{nil}(a), f, g, \star) &= \star \\
\mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\ldots, \mathsf{non\text{-}ind}(K, \gamma), f, g, \langle k, x\rangle) &= \mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\gamma(k), f, g, x) \\
\mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\ldots, \mathsf{A\text{-}ind}(K, \gamma), \ldots, f, g, \langle j, x\rangle) &= \langle f \circ j, \mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\ldots, \gamma, \ldots, f, g, x)\rangle \\
\mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\ldots, \mathsf{B\text{-}ind}(K, h, \gamma), \ldots, f, g, \langle j, x\rangle) &= \\
&\langle \lambda k.\, g(\overline{\mathsf{rep}_\mathrm{A}}(\ldots, h(k)), j(k)), \mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\ldots, \gamma, \ldots, f, g, x)\rangle
\end{aligned}$$

We define

$$\begin{aligned}
\mathsf{IH}^0_\mathrm{A}(\gamma_\mathrm{A}, P, Q) &: \mathsf{Arg}^0_\mathrm{A}(\gamma_\mathrm{A}, X, Y) \to \mathsf{Set} \\
\mathsf{IH}^0_\mathrm{B}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, P, Q) &: \mathsf{Arg}^0_\mathrm{B}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, X, Y, \mathsf{intro}_\mathrm{A}) \to \mathsf{Set} \\
\mathsf{map}^0_{\mathsf{IH}_\mathrm{A}}(\gamma_\mathrm{A}, f, g) &: (x : \mathsf{Arg}^0_\mathrm{A}(\gamma_\mathrm{A}, X, Y)) \to \mathsf{IH}^0_\mathrm{A}(\gamma_\mathrm{A}, P, Q, x) \\
\mathsf{map}^0_{\mathsf{IH}_\mathrm{B}}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, f, g) &: (x : \mathsf{Arg}^0_\mathrm{B}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, X, Y, \mathsf{intro}_\mathrm{A})) \to \mathsf{IH}^0_\mathrm{B}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, P, Q, x)
\end{aligned}$$

for $\gamma_\mathrm{A} : \mathsf{SP}^0_\mathrm{A}$ and $\gamma_\mathrm{B} : \mathsf{SP}^0_\mathrm{B}(\gamma_\mathrm{A})$ by

$$\begin{aligned}
\mathsf{IH}^0_\mathrm{A}(\gamma_\mathrm{A}, P, Q) &:= \mathsf{IH}_\mathrm{A}(\mathbf{0}, \gamma, X, Y, !(), P, Q) \\
\mathsf{IH}^0_\mathrm{B}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, P, Q) &:= \mathsf{IH}_\mathrm{B}(\mathbf{0}, \mathbf{0}, \gamma, X, Y, \mathsf{intro}_\mathrm{A}, !, !, !, P, Q) \\
\mathsf{map}^0_{\mathsf{IH}_\mathrm{A}}(\gamma_\mathrm{A}, f, g) &:= \mathsf{map}_{\mathsf{IH}_\mathrm{A}}(\mathbf{0}, \gamma, X, Y, !, P, Q, f, g, x) \\
\mathsf{map}^0_{\mathsf{IH}_\mathrm{B}}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, f, g) &:= \mathsf{map}_{\mathsf{IH}_\mathrm{B}}(\mathbf{0}, \mathbf{0}, \gamma_\mathrm{A}, X, Y, \mathsf{intro}_\mathrm{A}, !, !, !, P, Q, \gamma_\mathrm{B}, f, g)
\end{aligned}$$

as usual. We can now present the simple elimination principle for the inductive-inductive definition given by the codes γ_A, γ_B. We suppress the common premises $\gamma_\mathrm{A} : \mathsf{SP}^0_\mathrm{A}$, $\gamma_\mathrm{B} : \mathsf{SP}^0_\mathrm{B}(\gamma_\mathrm{A})$.

Axioms 3.16 (Simple elimination rules) Simple elimination rule for $A_{\gamma_\mathrm{A}, \gamma_\mathrm{B}}$:

$$\frac{\begin{array}{cc} P : X \to \mathsf{Set} & f : (x : \mathsf{Arg}^0_\mathrm{A}(\gamma_\mathrm{A}, A_{\gamma_\mathrm{A}, \gamma_\mathrm{B}}, B_{\gamma_\mathrm{A} \cdot \gamma_\mathrm{B}})) \to \mathsf{IH}^0_\mathrm{A}(\gamma_\mathrm{A}, P, Q, x) \to P(\mathsf{intro}_{A_{\gamma_\mathrm{A} \cdot \gamma_\mathrm{B}}}(x)) \\ Q : (x : X) \to Y(x) \to \mathsf{Set} & g : (y : \mathsf{Arg}^0_\mathrm{B}(\gamma_\mathrm{A}, \gamma_\mathrm{B}, A_{\gamma_\mathrm{A} \cdot \gamma_\mathrm{B}}; B_{\gamma_\mathrm{A}, \gamma_\mathrm{B}}, \mathsf{intro}_{A_{\gamma_\mathrm{A} \cdot \gamma_\mathrm{B}}})) \to \mathsf{IH}^0_\mathrm{B}(\gamma_\mathrm{B}; P, Q, y) \to Q(\mathsf{Index}^0_\mathrm{B}(\gamma_\mathrm{B}, y). \mathsf{intro}_{B_{\gamma_\mathrm{A}, \gamma_\mathrm{B}}}(y)) \end{array}}{\mathsf{elim}_{A_{\gamma_\mathrm{A}, \gamma_\mathrm{B}}}(P, Q, f, g) : (x : A_{\gamma_\mathrm{A}, \gamma_\mathrm{B}}) \to P(x)}$$

Simple elimination rule for $B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{\begin{array}{cc} P : X \to \mathsf{Set} & f : (x : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}.\, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}})) \to \mathsf{IH}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}.\, P, Q, x) \to P(\mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(x)) \\ Q : (x : X) \to Y(x) \to \mathsf{Set} & g : (y : \mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}})) \to \mathsf{IH}^0_{\mathrm{B}}(\gamma_{\mathrm{B}}, P, Q, y) \to Q(\mathsf{Index}^0_{\mathrm{B}}(\gamma_{\mathrm{B}}, y), \mathsf{intro}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(y)) \end{array}}{\mathsf{elim}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g) : (x : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}) \to (y : B(x)) \to Q(x, y)}$$

Computation rules:

$$\begin{aligned} &\mathsf{elim}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(x)) = \\ &\qquad f(x, \mathsf{map}^0_{\mathsf{IH}_{\mathrm{A}}}(\gamma_{\mathrm{A}}, \mathsf{elim}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g), \mathsf{elim}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g), x)) \\ &\mathsf{elim}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g, \mathsf{Index}^0_{\mathrm{B}}(\gamma_{\mathrm{B}}, y), \mathsf{intro}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(y)) = \\ &\qquad g(y, \mathsf{map}^0_{\mathsf{IH}_{\mathrm{B}}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, \mathsf{elim}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g), \mathsf{elim}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(P, Q, f, g), y)) \quad \blacksquare \end{aligned}$$

3.3 Summary and discussion

By extending the type theory given in Chapter 2, we have given a finite axiomatisation of inductive-inductive definitions. The axiomatisation is given as a schema of inductive-inductive definitions, represented by the type of their constructors, but *internalised* in Type Theory; we introduce a universe of codes for sets defined inductive-inductively, together with their decoding. The power of the theory then lies in the rules which says that each code has an associated constructor, and satisfies an elimination rule.

Equality, and how to avoid it In the proof of Lemma 3.9, we constructed a function of type

$$\mathsf{Arg}_{\mathrm{A}}(\ldots, \mathsf{rep}_{\mathrm{X}}, \ldots) \to \mathsf{Arg}_{\mathrm{A}}(\ldots, \mathsf{rep}'_{\mathrm{X}}, \ldots) \ , \tag{3.1}$$

given that $\mathsf{rep}'_{\mathrm{X}} = f \circ \mathsf{rep}_{\mathrm{X}}$. Why did we not simply construct a function of type

$$\mathsf{Arg}_{\mathrm{A}}(\ldots, \mathsf{rep}_{\mathrm{X}}, \ldots) \to \mathsf{Arg}_{\mathrm{A}}(\ldots, f \circ \mathsf{rep}_{\mathrm{X}}, \ldots)$$

instead? The reason is that $\mathsf{rep}'_{\mathrm{X}} = f \circ \mathsf{rep}_{\mathrm{X}}$ is actually too strong a requirement – we only needed $\mathsf{rep}'_{\mathrm{X}}$ and $f \circ \mathsf{rep}_{\mathrm{X}}$ to be *pointwise* equal, which is lucky, since this is all they are in certain recursive calls. Thus, we can define a function like in (3.1), as long as we can maintain the invariant that $\mathsf{rep}'_{\mathrm{X}}(x) = f(\mathsf{rep}_{\mathrm{X}}(x))$ for all $x : X_{\mathrm{ref}}$, even as X_{ref} grows.

In impredicative Type Theory, equality can be defined as Leibniz equality: two elements $x, y : A$ are equal if they satisfy the same properties, i.e. if

$$(P : A \to \mathsf{Set}) \to P(x) \to P(y)$$

is inhabited. This is not possible in predicative Martin-Löf Type Theory, as quantification over all $P : A \to \mathsf{Set}$ as a small type is not possible. However, if we already know that we only want to use x and y in a finite number of ways, we can instantiate P with those specific properties that we want to use, and eliminate a use of the identity type. In other words, instead of proving $p : x \equiv_A y$ and later using $\mathsf{subst}(B, p, x)$, we can prove $\tilde{p}_B : B(x) \to B(y)$ directly (if the proof of p was by refl, $\tilde{p}_B$ will be the identity function). This is what we do in the proof of Lemma 3.9, so that the identity type can become an instance of our theory instead of a prerequisite.

Implementing inductive-inductive definitions The theory we have presented should lend itself quite well for implementation; on top of a "normal" type theory, certain constants are postulated with a certain reduction behaviour, but always in a type-safe and sensible way. Indeed, we have not done so, but SP_A and SP_B can be considered to be (large) inductive definitions, in which case Arg_A, Arg_B etc can be considered to be defined by recursion over the codes. This way, the theory can be formalised in Agda (see Appendix A).

Of course, for an actual implementation, the user would not work directly with the codes in SP_A and SP_B, but would rather write data type declarations that would be elaborated to codes in the core Type Theory [Dagand and McBride, 2013]. It would be interesting to see how far this idea can be taken.

CHAPTER 4

A categorical characterisation

Contents

In this chapter, we seek a more abstract characterisation of inductive-inductive definitions. In the spirit of initial algebra semantics, we will characterise inductive-inductive definitions as initial objects in a category of "algebras". First we develop a generic framework for elimination rules which makes it possible to abstract away from the details when proving the equivalence of initiality in the category and the standard elimination rules. We then instantiate the framework to inductive-inductive definitions (with the general elimination rules) by considering an appropriate category. Because of the categorical setting, we work in extensional type theory in this chapter.

Parts of this chapter have appeared in the proceedings of CALCO 2011 [Altenkirch, Morris, Nordvall Forsberg, and Setzer, 2011].

4.1 Inductive-inductive definitions as dialgebras

Within the paradigm of initial algebra semantics [Goguen et al., 1977], a data type is modelled as the carrier of the initial algebra of a functor F. In more detail, let $\mathbb{C}$ be a category whose objects we think of as data types, and let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor on $\mathbb{C}$. An *F-algebra* is a pair (X, f) where X is an object of $\mathbb{C}$ and $f : F(X) \to X$. We call X the *carrier* of the algebra.

For any endofunctor F, the collection of F-algebras itself forms a category Alg_F of F-algebras. A morphism from (X, f) to (Y, g) is a map $h : X \to Y$ in $\mathbb{C}$ such that the following diagram commutes:

$$\begin{tikzcd}
F(X) \arrow[r, "f"] \arrow[d, "F(h)"'] & X \arrow[d, "h"] \\
F(Y) \arrow[r, "g"'] & Y
\end{tikzcd}$$

The initial F-algebra $(\mu F, \mathsf{in}_F)$ is the initial object in this category. As all initial objects, when it exists, it is unique up to isomorphism. The object μF is the interpretation of the data type described by F, while the morphism $\mathsf{in}_F : F(\mu F) \to \mu F$ interprets its constructors. We call F the *pattern functor* for the data type μF. Initiality ensures that, given any F-algebra $g : F(X) \to X$, there is a unique F-algebra homomorphism $\mathsf{fold}_F\ g$ from the initial algebra $(\mu F, \mathsf{in}_F)$ to that algebra. This is the semantic counterpart of the elimination rule for μF.

This gives a principled and expressive formalism for dealing with the semantics of simply typed data types. However, when trying to use initial algebra semantics to model inductive-inductive definitions, we run into two problems: (i) it is not enough to consider endofunctors $F : \mathbb{C} \to \mathbb{C}$, and (ii) we need to talk about dependent function spaces. The first problem is particular for inductive-inductive definitions. We will see how it arises, and discuss a solution to it in Section 4.1.1. The second problem is common to initial algebra approaches for dependent type theory in general, and a solution has been rediscovered in slightly different settings multiple times for many different systems; Closest to our own is Dybjer and Setzer's solution for induction-recursion [Dybjer and Setzer, 2003]. We develop a framework that can be instantiated to yield these different instances in Section 4.2, also taking the first problem (i) into account.

4.1.1 Dialgebras

One could imagine that inductive-inductive definitions could be described by functors mapping families of sets to families of sets (similar to the situation for induction-recursion [Dybjer and Setzer, 2003]), but this fails to take into account that the constructors for B should be able to refer to the constructors for A. We have seen in Chapter 3 that the constructor for B can be described by an operation

$$\mathsf{Arg_B} : (A : \mathsf{Set})(B : A \to \mathsf{Set})(c : \mathsf{Arg_A}(A, B) \to A) \to \mathsf{Arg_A}(A, B) \to \mathsf{Set}$$

where $c : \mathsf{Arg_A}(A, B) \to A$ refers to the already defined constructor for A. However, $(\mathsf{Arg_A}, \mathsf{Arg_B})$ is then no longer an endofunctor. We will model the constructor for B as (the second component of) a morphism $(c, d) : \mathsf{Arg}(A, B, c) \to (A, B)$ between families of sets. Recall from Definition 3.6 that if $\mathbb{D}$ is a category, then $\mathsf{Fam}\,\mathbb{D}$ is the category which has as objects pairs (A, B), where $A : \mathsf{Set}$ and $B : A \to \mathbb{D}$. A morphism from (A, B) to (A', B') is a pair (f, g) where $f : A \to A'$ and $g : (x : A) \to B(x) \to B'(f(x))$.

Note that there is a forgetful functor $U : \mathsf{Fam}\,\mathbb{D} \to \mathsf{Set}$ sending (A, B) to A and (f, g) to f, which we call the *index set functor*. We are interested in the situation where $\mathbb{D} = \mathsf{Set}$. Now, $c : \mathsf{Arg_A}(A, B) \to A$ is not an $\mathsf{Arg_A}$-algebra, since $\mathsf{Arg_A} : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ is not an endofunctor. However, we have $c : \mathsf{Arg}_{\mathsf{A}}(A, B) \to U(A, B)$. This means that c is a $(\mathsf{Arg}_{\mathsf{A}}, U)$-dialgebra [Hagino, 1987]:

Definition 4.1 Let $F, G : \mathbb{C} \to \mathbb{D}$ be functors. The category $\mathsf{Dialg}(F, G)$ has as objects pairs (X, f) where X is an object in $\mathbb{C}$ and $f : F(X) \to G(X)$. A morphism from (X, f) to (Y, g) is a morphism $h : X \to Y$ in $\mathbb{C}$ such that the following diagram in $\mathbb{D}$ commutes:

$$\begin{tikzcd} F(X) \arrow[r, "f"] \arrow[d, "F(h)"'] & G(X) \arrow[d, "G(h)"] \\ F(Y) \arrow[r, "g"'] & G(Y) \end{tikzcd} \tag{4.1}$$

■

Let $F, G : \mathbb{C} \to \mathbb{D}$. There is a forgetful functor $V : \mathsf{Dialg}(F, G) \to \mathbb{C}$ defined by $V(A, f) = A$. Dialgebras are called subequalisers by Lambek [1970], and are a special case of inserters [Kelly, 1989] in the 2-category of categories. If we choose G to be the identity functor $\mathsf{Id} : \mathbb{C} \to \mathbb{C}$, we recover the concept of an F-algebra.

Putting things together, we will model the constructor for A as a morphism $c : \mathsf{Arg}_{\mathsf{A}}(A, B) \to U(A, B)$ in Set, i.e. as a function $c : \mathsf{Arg}_{\mathsf{A}}(A, B) \to A$, and the constructor for B as the second component of a morphism $(c, d) : \mathsf{Arg}(A, B, c) \to V(A, B, c)$ in $\mathsf{Fam}(\mathsf{Set})$, i.e. as the second component of a morphism $(c, d) : \mathsf{Arg}(A, B, c) \to (A, B)$. Thus, we see that the data needed to describe (A, B) as inductively generated with constructors c, d are the functors $\mathsf{Arg}_{\mathsf{A}}$ and Arg. However, we must also make sure that the first component of Arg coincides with $\mathsf{Arg}_{\mathsf{A}}$, i.e. that $U \circ \mathsf{Arg} = \mathsf{Arg}_{\mathsf{A}} \circ V$. Each code in the axiomatisation in Chapter 3 gives rise to such functors:

Proposition 4.2 Each code $\gamma = (\gamma_A, \gamma_B)$ for an inductive-inductive definition induces two functors

$$\mathsf{Arg}_{\mathsf{A}} : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set} \qquad \mathsf{Arg} : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$$

defined by

$$\mathsf{Arg}_{\mathsf{A}}(A, B) := \mathsf{Arg}^0_{\mathsf{A}}(\gamma_A, A, B)$$

and

$$\begin{aligned} &\mathsf{Arg}(A, B, c) := \\ &\quad (\mathsf{Arg}_{\mathsf{A}}(A, B), \lambda x. \{\, y{:}\mathsf{Arg}^0_{\mathsf{B}}(\gamma_A, \gamma_B, A, B, c) \mid c(x) = \mathsf{Index}^0_{\mathsf{B}}(\gamma_{\mathsf{A}}, A, B, c, \gamma_{\mathsf{B}}, y)\}) \ . \end{aligned}$$

Note that $U \circ \mathsf{Arg} = \mathsf{Arg}_{\mathsf{A}} \circ V$.

Proof. We have already seen that $\mathsf{Arg}_{\mathsf{A}}$ is functorial in Lemma 3.9. Similarly, Arg can be proven to be functorial by induction over γ_{B}, making crucial use of extensionality and the fact that morphisms in $\mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U)$ make diagrams of the form (4.1) commute. □

Remark 4.3 If we have two functors Arg_A, Arg as in Proposition 4.2, that is

$$\begin{aligned}\mathsf{Arg}_\mathsf{A} &: \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set} \\ \mathsf{Arg} &: \mathsf{Dialg}(\mathsf{Arg}_\mathsf{A}, U) \to \mathsf{Fam}(\mathsf{Set})\end{aligned}$$

with $U \circ \mathsf{Arg} = \mathsf{Arg}_\mathsf{A} \circ V$, then the first functor is determined as the first component of the second, and we often write such a pair as $\mathsf{Arg} = (\mathsf{Arg}_\mathsf{A}, \mathsf{Arg}_\mathsf{B})$ where

$$\mathsf{Arg}_\mathsf{B} : (A : \mathsf{Set})(B : A \to \mathsf{Set})(c : \mathsf{Arg}_\mathsf{A}(A,B) \to A) \to \mathsf{Arg}_\mathsf{A}(A,B) \to \mathsf{Set} \ .$$

Example 4.4 (Contexts and types) The inductive-inductive definition of $\mathsf{Ctxt} : \mathsf{Set}$ and $\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set}$ is given by

$$\begin{aligned}\mathsf{Arg}_\mathsf{Ctxt}(A,B) &= \mathbf{1} + \Sigma\,\Gamma{:}A.\,B(\Gamma) \\ \mathsf{Arg}_\mathsf{Ty}(A,B,c,x) &= \mathbf{1} + \Sigma\,\sigma{:}B(c(x)).\,B(c(\mathsf{inr}(c(x),\sigma))) \ .\end{aligned}$$

For $\mathsf{Arg}_\mathsf{Ctxt}$, the left summand $\mathbf{1}$ corresponds to the constructor ε taking no arguments, and the right summand $\Sigma\,\Gamma{:}A.\,B(\Gamma)$ corresponds to $\rhd$'s two arguments $\Gamma : \mathsf{Ctxt}$ and $\sigma : \mathsf{Ty}(\Gamma)$. Similar considerations apply to Arg_Ty. ■

Example 4.5 (Sorted lists) The sorted list example does not fit into our framework, since $\le_\mathsf{L}$: $(\mathbb{N} \times \mathsf{SortedList}) \to \mathsf{Set}$ is indexed by $\mathbb{N} \times \mathsf{SortedList}$ and not simply $\mathsf{SortedList}$. It is however straightforward to generalise the construction to include this example as well: instead of considering ordinary families, consider "$\mathbb{N} \times A$-indexed" families (A,B) where A is a set and $B : (\mathbb{N} \times A) \to \mathsf{Set}$. The inductive-inductive definition of $\mathsf{SortedList} : \mathsf{Set}$ and $\le_\mathsf{L}$: $(\mathbb{N} \times \mathsf{SortedList}) \to \mathsf{Set}$ is then given by

$$\begin{aligned}\mathsf{Arg}_\mathsf{SList}(A,B) &= \mathbf{1} + (\Sigma\,n{:}\mathbb{N}.\,\Sigma\,\ell{:}A.\,B(n,\ell)) \\ \mathsf{Arg}_{\le_\mathsf{L}}(A,B,c,m,\mathsf{inl}(\star)) &= \mathbf{1} \\ \mathsf{Arg}_{\le_\mathsf{L}}(A,B,c,m,\mathsf{inr}(\langle n,\ell,p\rangle)) &= \Sigma\,m \le n.\,B(m,\ell) \ .\end{aligned}$$

For ease of presentation, we will only consider ordinary families of sets in this chapter, but will extend the theory to cover this example as well in Section 6.2. ■

4.1.2 A category for inductive-inductive definitions

Given $\mathsf{Arg} = (\mathsf{Arg}_\mathsf{A}, \mathsf{Arg}_\mathsf{B})$ representing an inductive-inductive definition, we will now construct a category $\mathbb{E}_\mathsf{Arg}$ whose initial object (if it exists) is the intended interpretation of the inductive-inductive definition. Figure 4.1 summarises the functors and categories involved (U, V and W are all forgetful functors).

One might think that the category we are looking for is $\mathsf{Dialg}(\mathsf{Arg}, V)$, where $V : \mathsf{Dialg}(\mathsf{Arg}_\mathsf{A}, U) \to \mathsf{Fam}(\mathsf{Set})$ is the forgetful functor. $\mathsf{Dialg}(\mathsf{Arg}, V)$ has objects $(A, B, c, (d_0, d_1))$, where $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$, $c : \mathsf{Arg}_\mathsf{A}(A,B) \to A$ and $(d_0, d_1) : \mathsf{Arg}(A,B,c) \to (A,B)$. The function $d_0 : \mathsf{Arg}_\mathsf{A}(A,B) \to A$ looks like the constructor for A that we want, but

$$d_1 : (x : \mathsf{Arg}_\mathsf{A}(A,B)) \to \mathsf{Arg}_\mathsf{B}(A,B,c,x) \to B(d_0(x))$$

$$\begin{tikzcd}
\mathsf{Set} & \mathsf{Fam(Set)} \arrow[l, bend right, "\mathsf{Arg_A}"'] \arrow[l, bend left, "U"] & \mathsf{Dialg}(\mathsf{Arg_A}, U) \arrow[l, bend right, "\mathsf{Arg}"'] \arrow[l, bend left, "V"] & \mathsf{Dialg}(\mathsf{Arg}, V) \arrow[l, bend right, "{(V,U)}"'] \arrow[l, bend left, "W"] & \mathbb{E}_{\mathsf{Arg}} \arrow[l, hook]
\end{tikzcd}$$

Figure 4.1: The functors and categories involved.

does not have the type we expect of the constructor for B; it would, if only c and d_0 were the same! To this end, we will consider the equaliser of the forgetful functor

$$W : \mathsf{Dialg}(\mathsf{Arg}, V) \to \mathsf{Dialg}(\mathsf{Arg_A}, U)$$

defined by $W(A, B, c, (d_0, d_1)) = (A, B, c)$, and the functor (V, U) defined by

$$\begin{aligned}
(V,U)(A,B,c,(d_0,d_1)) &:= (V(A,B,c), U(d_0,d_1)) = (A,B,d_0)\\
(V,U)(f,g) &:= (f,g)
\end{aligned}$$

Note that $U(d_0, d_1) : U(\mathsf{Arg}(A, B, c)) \to U(V(A, B, c))$ but $U \circ \mathsf{Arg} = \mathsf{Arg_A} \circ V$, so that $U(d_0, d_1) : \mathsf{Arg_A}(V(A, B, c)) \to U(V(A, B, c))$. In other words, $(V(A, B, c), U(d_0, d_1))$ is an object in $\mathsf{Dialg}(\mathsf{Arg_A}, U)$. Hence (V, U) really is a functor from $\mathsf{Dialg}(\mathsf{Arg}, V)$ to $\mathsf{Dialg}(\mathsf{Arg_A}, U)$.

Definition 4.6 For $\mathsf{Arg} = (\mathsf{Arg_A}, \mathsf{Arg_B})$ representing an inductive-inductive definition, let $\mathbb{E}_{\mathsf{Arg}}$ be the underlying category of the equaliser of (V, U) and the forgetful functor $W : \mathsf{Dialg}(\mathsf{Arg}, V) \to \mathsf{Dialg}(\mathsf{Arg_A}, U)$. ◻

Explicitly, the category $\mathbb{E}_{\mathsf{Arg}}$ has

- Objects (A, B, c, d), where $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$, $c : \mathsf{Arg_A}(A, B) \to A$, $d : (x : \mathsf{Arg_A}(A, B)) \to \mathsf{Arg_B}(A, B, c, x) \to B(c(x))$.
- Morphisms from (A, B, c, d) to (A', B', c', d') are morphisms $(f, g) : (A, B, c) \Rightarrow_{\mathsf{Dialg}(\mathsf{Arg_A}, U)} (A', B', c')$ such that in addition
$$g(c(x), d(x, y)) = d'(\mathsf{Arg_A}(f, g)(x), \mathsf{Arg_B}(f, g)(x, y)) \ .$$

Example 4.7 Consider the functors $\mathsf{Arg_{Ctxt}}$, $\mathsf{Arg_{Ty}}$ from Example 4.4:

$$\begin{aligned}
\mathsf{Arg_{Ctxt}}(A,B) &= \mathbf{1} + \Sigma\Gamma{:}A.\, B(\Gamma)\\
\mathsf{Arg_{Ty}}(A,B,c,x) &= \mathbf{1} + \Sigma\sigma{:}B(c(x)).\, B(c(\mathsf{inr}(c(x),\sigma))) \ .
\end{aligned}$$

An object in $\mathbb{E}_{(\mathsf{Arg_{Ctxt}}, \mathsf{Arg_{Ty}})}$ consists of $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$ and morphisms $c = [\varepsilon_{A,B}, \rhd_{A,B}]$ and $d = \lambda\Gamma.\, [\iota_{A,B}(\Gamma), \Pi_{A,B}(\Gamma)]$ which can be split up into[1]

$$\begin{aligned}
&\varepsilon_{A,B} : \mathbf{1} \to A \ , \qquad\qquad \rhd_{A,B} : \big((\Gamma : A) \times B(\Gamma)\big) \to A \ ,\\
&\iota_{A,B} : (\Gamma : \mathsf{Arg_{Ctxt}}(A,B)) \to \mathbf{1} \to B(c(\Gamma)) \ ,\\
&\Pi_{A,B} : (\Gamma : \mathsf{Arg_{Ctxt}}(A,B)) \to \big((\sigma : B(c(\Gamma))) \times (\tau : B(\rhd_{A,B}(c(\Gamma),\sigma)))\big) \to B(c(\Gamma)) \ . \ \blacksquare
\end{aligned}$$

[1] Notice that $\iota_{A,B} : (\Gamma : \mathsf{Arg_{Ctxt}}(A, B)) \to \ldots$ and not $\iota_{A,B} : (\Gamma : A) \to \ldots$ as one would maybe expect. There is no difference for initial A, as we have $\mathsf{Arg_{Ctxt}}(A, B) \cong A$ by (a variant of) Lambek's Lemma.

In Section 6.2, we will generalise the current construction to the simultaneous definition of $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$, $C : (x : A) \to B(x) \to \mathsf{Set}$, ...by extending the construction hinted at in Figure 4.1.

The intended interpretation of the inductive-inductive definition given by $\mathsf{Arg} = (\mathsf{Arg}_\mathsf{A}, \mathsf{Arg}_\mathsf{B})$ is the initial object in $\mathbb{E}_\mathsf{Arg}$. Depending on the meta-theory, this might of course not exist. We will show that it does if and only if an eliminator for the inductive-inductive definition exists. To avoid the messy details, we will prove the statement for a more abstract notion of data types in Section 4.2.

Remark 4.8 We need to pass from $\mathsf{Dialg}(\mathsf{Arg}, V)$ to the equaliser category $\mathbb{E}_\mathsf{Arg}$ since the objects in $\mathsf{Dialg}(\mathsf{Arg}, V)$ are not quite of the right shape for inductive-inductive definitions, which we intend to interpret as initial objects in the appropriate category. However, since $\mathbb{E}_\mathsf{Arg}$ is a subcategory of $\mathsf{Dialg}(\mathsf{Arg}, V)$, we could hope to prove that the initial object of $\mathsf{Dialg}(\mathsf{Arg}, V)$ already lives in $\mathbb{E}_\mathsf{Arg}$, in order to simplify the construction. I have not succeeded in doing so.

4.1.2.1 How to exploit initiality: an example

Let us consider an example of how to use initiality to derive a program dealing with the contexts and types from Example 4.7. Suppose that we want to define a concatenation $+\!\!+ : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Ctxt}$ of contexts – such an operation could be useful to formulate more general formation rules, such as:

$$\frac{\sigma : \mathsf{Ty}(\Gamma) \qquad \tau : \mathsf{Ty}(\Delta)}{\sigma \times \tau : \mathsf{Ty}(\Gamma +\!\!+ \Delta)}$$

Such an operation should satisfy the equations

$$\begin{array}{ccccc} \Delta & +\!\!+ & \varepsilon & = & \Delta \\ \Delta & +\!\!+ & (\Gamma \rhd \sigma) & = & (\Delta +\!\!+ \Gamma) \rhd (\mathsf{wk}_\Gamma(\sigma, \Delta)) \end{array} ,$$

where $\mathsf{wk} : (\Gamma : \mathsf{Ctxt}) \to (\sigma : \mathsf{Ty}(\Gamma)) \to (\Delta : \mathsf{Ctxt}) \to \mathsf{Ty}(\Delta +\!\!+ \Gamma)$ is a weakening operation, i.e. if $\sigma : \mathsf{Ty}(\Gamma)$, then $\mathsf{wk}_\Gamma(\sigma, \Delta) : \mathsf{Ty}(\Delta +\!\!+ \Gamma)$. A moment's thought should convince us that we want wk to satisfy

$$\begin{aligned} \mathsf{wk}_\Gamma(\iota_\Gamma, \Delta) &= \iota_{\Delta +\!\!+ \Gamma} \\ \mathsf{wk}_\Gamma(\Pi_\Gamma(\sigma, \tau), \Delta) &= \Pi_{\Delta +\!\!+ \Gamma}(\mathsf{wk}_\Gamma(\sigma, \Delta), \mathsf{wk}_{\Gamma \rhd \sigma}(\tau, \Delta)) \end{aligned} .$$

Our hope is now to exploit the initiality of $(\mathsf{Ctxt}, \mathsf{Ty})$ to get such operations. Recall from Example 4.4 that Ctxt, Ty are the underlying sets for the inductive-inductive definition given by the functors

$$\begin{aligned} \mathsf{Arg}_\mathsf{Ctxt}(A, B) &= \mathbf{1} + \Sigma\Gamma{:}A.\, B(\Gamma) \\ \mathsf{Arg}_\mathsf{Ty}(A, B, c, x) &= \mathbf{1} + (\Sigma\sigma{:}B(c(x)).\, \tau{:}B(c(\mathsf{inr}(c(x), \sigma)))) \end{aligned} .$$

From the types of

$$\begin{aligned} +\!\!+ &: \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Ctxt} \\ \mathsf{wk} &: (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \to (\Delta : \mathsf{Ctxt}) \to \mathsf{Ty}(\Delta +\!\!+ \Gamma) \ , \end{aligned}$$

we see that if we can equip (A, B) where $A := \mathsf{Ctxt} \to \mathsf{Ctxt}$ and $B(f) := (\Delta : \mathsf{Ctxt}) \to \mathsf{Ty}(f(\Delta))$ with an $(\mathsf{Arg}_{\mathsf{Ctxt}}, \mathsf{Arg}_{\mathsf{Ty}})$ structure, initiality will give us functions of the right type. Of course, we must choose the right structure so that our equations will be satisfied:

$$\begin{aligned} &\mathsf{in}_A : \mathsf{Arg}_{\mathsf{Ctxt}}(A, B) \to A \\ &\mathsf{in}_A(\mathsf{inl}(\star)) &&= \lambda\Delta.\,\Delta \\ &\mathsf{in}_A(\mathsf{inr}(\langle f, g\rangle)) &&= \lambda\Delta.\,(f(\Delta) \rhd g(\Delta)) \ , \end{aligned}$$

$$\begin{aligned} &\mathsf{in}_B : (x : \mathsf{Arg}_{\mathsf{Ctxt}}(A, B)) \to \mathsf{Arg}_{\mathsf{Ty}}(A, B, \mathsf{in}_A, x) \to B(\mathsf{in}_A(x)) \\ &\mathsf{in}_B(\Delta, \mathsf{inl}(\star)) &&= \lambda\Gamma.\,\iota_{\mathsf{in}_A(\Delta)(\Gamma)} \\ &\mathsf{in}_B(\Delta, \mathsf{inr}(\langle g, h\rangle)) &&= \lambda\Gamma.\,\Pi_{\mathsf{in}_A(\Delta)(\Gamma)}(g(\Gamma), h(\Gamma)) \ . \end{aligned}$$

Since $(A, B, \mathsf{in}_A, \mathsf{in}_B)$ is an object in $\mathbb{E}_{\mathsf{Arg}}$, initiality gives us a morphism $(+\!\!+, \mathsf{wk}) : (\mathsf{Ctxt}, \mathsf{Ty}) \to (A, B)$ such that

$$(+\!\!+, \mathsf{wk}) \circ ([\varepsilon, \rhd], [\iota, \Pi]) = (\mathsf{in}_A, \mathsf{in}_B) \circ (\mathsf{Arg}_{\mathsf{Ctxt}}, \mathsf{Arg}_{\mathsf{Ty}})(+\!\!+, \mathsf{wk}) \ .$$

In particular, this means that

$$\begin{aligned} +\!\!+(\varepsilon) &= \mathsf{in}_A(\mathsf{Arg}_{\mathsf{Ctxt}}(+\!\!+, \mathsf{wk})(\mathsf{inl}(\star))) = \mathsf{in}_A(\mathsf{inl}(\star)) = \lambda\Delta.\,\Delta \\ +\!\!+(\Gamma \rhd \sigma) &= \mathsf{in}_A(\mathsf{Arg}_{\mathsf{Ctxt}}(+\!\!+, \mathsf{wk})(\mathsf{inr}(\langle \Gamma, \sigma\rangle))) = \mathsf{in}_A(\mathsf{inr}(\langle +\!\!+(\Gamma), \mathsf{wk}(\Gamma, \sigma)\rangle)) \\ &= \lambda\Delta.\ +\!\!+(\Gamma, \Delta) \rhd \mathsf{wk}(\Gamma, \sigma, \Delta) \ . \end{aligned}$$

Thus, we see that $\Delta +\!\!+ \varepsilon = \Delta$ and $\Delta +\!\!+ (\Gamma \rhd \sigma) = (\Delta +\!\!+ \Gamma) \rhd \mathsf{wk}_\Gamma(\sigma, \Delta)$ as required.[2] In the same way, the equations for the weakening operation hold.

4.2 A framework for generic elimination rules

We now prove the equivalence between being initial and having an eliminator. To make the size of the equations we have to juggle around bearable, we will abstract away from the particular instance of inductive-inductive definitions and consider a general framework of elimination rules for different kinds of "inductive-like" definitions. Our concrete result can then be reconstructed by instantiating the framework with the category introduced in Section 4.1.2.

[2] Actually, the order of the arguments is reversed, so we would have to define $\Delta +\!\!+' \Gamma := \Gamma +\!\!+ \Delta$.

4.2.1 Categories with Families

In order to come up with a framework for generic elimination rules, we try to isolate the different concepts involved in a dependent eliminator, in contrast to an iterator such as fold_F.

(i) We can model data types and non-dependent functions as objects and morphisms in a category $\mathbb{C}$. For instance, inductive definitions X : Set can be modelled as objects in Set, and I-indexed inductive families $X : I \to \mathrm{Set}$ as objects in Set^I.

(ii) We need to be able to talk about *predicates* on X (for instance, for an inductive definition X : Set, we would like to talk about $P : X \to \mathrm{Set}$, and for an inductive family $X : I \to \mathrm{Set}$, a predicate P on X should have type $P : (i : I) \to X(i) \to \mathrm{Set}$).

(iii) Given a predicate P on X, we want to be able to form a "sigma type" or *comprehension* $\Sigma_X P$ of the same standing as X – in other words, $\Sigma_X P$ should also be an object in $\mathbb{C}$ (for instance, an ordinary sigma type $\Sigma x{:}X.P(x)$ for an inductive definition X : Set, and an "indexed sigma type" $i \mapsto \Sigma x : X(i).P(i,x)$ for an inductive family $X : I \to \mathrm{Set}$).

(iv) Given a predicate P, we want to consider "dependent functions" $f : \Pi_X P$. Just as hom-sets in a category need not be represented internally by an exponential object, we do not need to demand that the collection of all such dependent functions is represented in $\mathbb{C}$. (For instance, an ordinary dependent function $f : (x : X) \to P(x)$ for an inductive definition X : Set, and "indexed dependant functions" $f : (i : I) \to (x : X(i)) \to P(i,x)$ for an inductive family $X : I \to \mathrm{Set}$).

This looks like the structure of a *Category with Families* [Dybjer, 1996; Hofmann, 1997]. Categories with Families were introduced by Dybjer [1996] as "uncategorical categorical models of Type Theory" with the aim of being a syntax-free presentation of Type Theory as close to the syntax as possible. Many similar models have been proposed, e.g. Cartmell's categories with attributes [Cartmell, 1978], Jacobs' comprehension categories [Jacobs, 1993] and Taylor's display map categories [Taylor, 1999].

Definition 4.9 A *Category with Families* (*CwF*) is given by

- A category $\mathbb{C}$ with a terminal object $\mathbf{1}$,
- A functor $F : \mathbb{C}^{\mathrm{op}} \to \mathsf{Fam}(\mathrm{Set})$. We write $F(\Gamma) = (\mathrm{Ty}(\Gamma), \lambda A.\,(\Gamma \vdash A))$ for the two components of F. For the morphism part, we introduce the notation $-[-]$ for both types and terms, i.e. if $f : \Delta \to \Gamma$ then $-[f] : \mathrm{Ty}(\Gamma) \to \mathrm{Ty}(\Delta)$ and for every $A \in \mathrm{Ty}(\Gamma)$ we have $-[f] : (\Gamma \vdash A) \to (\Delta \vdash A[f])$.
- For each object Γ in $\mathbb{C}$ and $A \in \mathrm{Ty}(\Gamma)$ an object $\Gamma \cdot A$ in $\mathbb{C}$, the *context comprehension* of Γ and A, together with a morphism $\mathbf{p}(A) : \Gamma \cdot A \to \Gamma$ (the*first projection*) and a term $\mathbf{q}_A \in (\Gamma \cdot A \vdash A[\mathbf{p}(A)])$ (the *second projection*) with the following universal property: for each $f : \Delta \to \Gamma$ and $M \in (\Delta \vdash A[f])$ there exists a unique morphism

$\theta = \langle f, M\rangle_A : \Delta \to \Gamma \cdot A$ such that $\mathbf{p}(A) \circ \theta = f$ and $\mathbf{q}_A[\theta] = M$. We write $\mathbf{p}$ and $\mathbf{q}$ for $\mathbf{p}(A)$ and $\mathbf{q}_A$ respectively if A can be inferred from context. ■

We note in passing that Type Theory can be interpreted in a Category with Families by interpreting contexts as objects in $\mathbb{C}$, types in context Γ as elements of $\mathrm{Ty}(\Gamma)$ and terms of type σ in context Γ as elements of $(\Gamma \vdash \sigma)$. That $F : \mathbb{C}^{\mathrm{op}} \to \mathsf{Fam}(\mathsf{Set})$ is a functor corresponds to substitution in types and terms. The comprehension $\Gamma \cdot \sigma$ models the extension of the context Γ with a fresh variable of type σ, with weakening $-[\mathbf{p}] : \mathrm{Ty}(\Gamma) \to \mathrm{Ty}(\gamma \cdot \sigma)$. The fresh variable is available as the term $\mathbf{q} \in (\Gamma \cdot \sigma \vdash \sigma[\mathbf{p}(\sigma)])$ in the extended context, with (weakened) type σ.

Example 4.10 (Set as a CwF) The category of sets becomes a Category with Families if we define[3]

$$\mathrm{Ty}(\Gamma) = \{A \mid A : \Gamma \to \mathsf{Set}\} \qquad (\Gamma \vdash A) = \prod_{\gamma \in \Gamma} A(\gamma)$$

For $f : \Delta \to \Gamma$, $A : \mathrm{Ty}(\Gamma)$, $h : (\Gamma \vdash A)$, we define

$$\begin{array}{ll} A[f] : \mathrm{Ty}(\Delta) = \{B \mid B : \Delta \to \mathsf{Set}\} & h[f] : (\Delta \vdash A[f]) = \prod_{\delta \in \Delta} A(f(\delta)) \\ A[f] = A \circ f & h[f] = h \circ f \end{array}$$

We define the context comprehension of Γ : Set and $A : \Gamma \to \mathsf{Set}$ as $\Gamma \cdot A = \sum_{\gamma \in \Gamma} A(\gamma)$. There are projections

$$\begin{array}{ll} \mathbf{p}(A) : \sum_{\gamma \in \Gamma} A(\gamma) \to \Gamma & \mathbf{q}_A \in (\Gamma \cdot A \vdash A[\mathbf{p}(A)]) = \prod_{\langle \gamma, s\rangle \in \Gamma \cdot A} A(\gamma) \\ \mathbf{p}(A)(\langle \gamma, s\rangle) = \gamma & \mathbf{q}_A(\langle \gamma, s\rangle) = s \end{array}$$

Finally, given $f : \Delta \to \Gamma$ and $M \in (\Delta \vdash A[f]) = \prod_{\delta \in \Delta} A(f(\delta))$, we define

$$\theta = \langle f, M\rangle_A : \Delta \to \Gamma \cdot A$$

by $\theta(\delta) = \langle f(\delta), M(\delta)\rangle$. We then have $\mathbf{p}(A) \circ \theta = f$ and $\mathbf{q}_A[\theta] = M$, and any other function satisfying these equations must be extensionally equal to θ, hence θ is unique. ■

Comparing our list of requirements from the beginning of the Section with Definition 4.9, we think of a Category with Families $\mathbb{C}$ in the following way:

(i) We model data types and non-dependent functions as objects and morphisms in $\mathbb{C}$.

(ii) We model predicates on X as elements of $\mathrm{Ty}(X)$.

[3] For size reasons, we should restrict $\mathrm{Ty}(\Gamma)$ to Γ-indexed families of *small* sets, that is, type-theoretically, use a universe (U, T) and define $\mathrm{Ty}(\Gamma) = \{A \mid A : \Gamma \to U\}$, and accordingly $(\Gamma \vdash A) = \prod_{\gamma \in \Gamma} T(A(\gamma)))$.

(iii) We model predicate comprehension $\Sigma_X P$ as context comprehension $X \cdot P$.

(iv) We model dependent functions $f : \Pi_X P$ as elements $f : (X \vdash P)$.

Note that we are *not* working in the model of type theory that the Category with Families describes. Instead we are shifting everything one level: contexts become types, types become type families. The terminal object in $\mathbb{C}$ (normally representing the empty context) plays less of a rôle for our purposes. In Example 4.10, we see how this perfectly matches the setting of ordinary inductive definitions. This is true also for e.g. indexed inductive definitions and, of course, inductive-inductive definitions.

We will repeatedly use the following lemma when working with Categories with Families.

Lemma 4.11

(i) Let $f : \Delta \to \Gamma$, $M \in (\Delta \vdash A[f])$, $h : \Theta \to \Delta$. Then $\langle f, M\rangle_A \circ h = \langle f \circ h, M[h]\rangle_{A[f]}$.

(ii) For every $M \in (\Gamma \vdash A)$, there is $\overline{M} : \Gamma \to \Gamma \cdot A$ such that $\mathsf{p}(A) \circ \overline{M} = \mathsf{id}$ and $\mathsf{q}_A[\overline{M}] = M$.

Proof.

(i) $\langle f, M\rangle_\sigma \circ h$ satisfies the universal property for $f \circ h$ and $M[h]$.

(ii) There is no choice but to define $\overline{M} := \langle \mathsf{id}, M\rangle_A$. □

4.2.2 A generic induction hypothesis type

In the axiomatisation of the elimination rules in Section 3.2.5, we defined types $\mathsf{IH_A}$, $\mathsf{IH_B}$ of induction hypothesis for the step functions of the eliminators. Our goal in this section is to come up with an abstract counterpart of the induction hypothesis type in the Categories with Families framework. In fact, the elimination rules we model will turn out to be the general elimination rules, and not just the simple ones discussed in Section 3.2.5.2. But first, let us take a step back and consider an eliminator for an inductive definition, i.e. for an F-algebra (A, f) where $F : \mathsf{Set} \to \mathsf{Set}$. Such an eliminator is of the form

$$\frac{P : A \to \mathsf{Set} \qquad \mathsf{step}_c : (x : F(A)) \to \Box_F(P, x) \to P(c(x))}{\mathsf{elim}_F(P, \mathsf{step}_c) : (x : A) \to P(x)}$$

where we have written $\Box_F(P) : F(A) \to \mathsf{Set}$ for the type of inductive hypothesis with respect to P; we have used the notation $\Box$ from modal logic, since $\Box_F(P, x)$ consists of proofs that P holds at all F-substructures of x. The rule says that if we can prove that $P(y)$ holds for elements $y = c(x)$ constructed with the constructor c (given that it already holds for the subelements of x by the induction hypothesis), then it holds for all elements of A.

We also expect a corresponding computation rule which tells us how $\mathsf{elim}_F(P, \mathsf{step}_c)$ behaves when applied to canonical elements:

$$\mathsf{elim}_F(P, \mathsf{step}_c, c(x)) = \mathsf{step}_c(x, \overline{F}(P, \mathsf{elim}(P, \mathsf{step}_c), x)) \ .$$

Here, $\overline{F}(P) : (f : (x : A) \to P(x)) \to (x : F(A)) \to \Box_F(P, x)$ takes care of recursive calls. We will discuss $\overline{F}$ in more detail in Section 4.2.3.

Example 4.12 Let $F(X) = \mathbf{1} + X$, i.e. F is the functor whose initial algebra is $(\mathbb{N}, [0, \mathsf{suc}])$. The type of induction hypothesis $\Box_{\lambda X.\, 1+X}$ should then satisfy

$$\Box_{\lambda X.\, 1+X}(P, \mathsf{inl}(\star)) \cong \mathbf{1} \qquad \Box_{\lambda X.\, 1+X}\ (P, \mathsf{inr}(n)) \cong P(n)$$

so that the eliminator for $(\mathbb{N}, [0, \mathsf{suc}])$ becomes

$$\dfrac{P : \mathbb{N} \to \mathsf{Set} \qquad \begin{array}{c} \mathsf{step}_0 : \mathbf{1} \to P(0) \\ \mathsf{step}_{\mathsf{suc}} : (n : \mathbb{N}) \to P(n) \to P(\mathsf{suc}(n)) \end{array}}{\mathsf{elim}_{\mathbf{1}+X}(P, \mathsf{step}_0, \mathsf{step}_{\mathsf{suc}}) : (x : \mathbb{N}) \to P(x)} \qquad \blacksquare$$

For polynomial functors F, $\Box_F$ can be defined inductively over the structure of F as is given in e.g. Dybjer and Setzer [2003]; Hermida and Jacobs [1998]. However, $\Box_F$ and $\overline{F}$ can be defined for any functor $F : \mathsf{Set} \to \mathsf{Set}$ by defining

$$\begin{aligned} \Box_F(P, x) &:= \{y : F(\Sigma\, z{:}A.\, P(z)) \mid F(\pi_0)(y) = x\} \\ \overline{F}(P, \mathsf{step}_c, x) &:= \langle F(\overline{\mathsf{step}_c})(x), \mathsf{refl}\rangle \ , \end{aligned} \tag{4.2}$$

where we have used the notation $\overline{\mathsf{step}_c} := \lambda y.\, \langle y, \mathsf{step}_c(y)\rangle$. Returning to Example 4.12, we see that indeed $\Box_{\lambda X.\, 1+X}(P, \mathsf{inl}(\star)) \cong \mathbf{1}$ and $\Box_{\lambda X..1+X}(P, \mathsf{inr}(n)) \cong P(n)$.

Lemma 4.13 Let $F : \mathsf{Set} \to \mathsf{Set}$ and let $\Box_F$ and $\overline{F}$ be defined as in (4.2).

(i) There is an isomorphism $\varphi : F(\Sigma\ A\ B) \xrightarrow{\cong} \Sigma\ F(A)\ (\Box_F(B))$ with $\pi_0 \circ \varphi = F(\pi_0)$.

(ii) For $g : (x : A) \to B(x)$, we have $\overline{\overline{F}(g)} = \varphi \circ F(\overline{g})$.

Proof.

(i) Define φ with type as above and $\psi : \Sigma\ (FA)\ (\Box_F B)) \to F(\Sigma\ A\ B)$ by

$$\varphi(y) = \langle F(\pi_0)(y), \langle y, \mathsf{refl}\rangle\rangle \ , \qquad \psi(\langle x, \langle y, p\rangle\rangle) = y.$$

Then $\psi(\varphi(y)) = \psi(\langle F(\pi_0)\, y, \langle y, \mathsf{refl}\rangle\rangle) = y$ and for every $\langle x, \langle y, p\rangle\rangle : \Sigma\ (FA)\ (\Box_F B))$, we have $x = F(\pi_0)(y)$ by p and $p = \mathsf{refl}$ by proof irrelevance, so that

$$\varphi(\psi(\langle x, \langle y, p\rangle\rangle)) = \varphi(y) = \langle F(\pi_0)(y), \langle y, \mathsf{refl}\rangle\rangle = \langle x, \langle y, p\rangle\rangle \ .$$

Hence $F(\Sigma\ A\ B) \cong \Sigma\ (FA)\ (\Box_F B))$.

(ii) By definition, $\overline{\overline{F}(g)}(x) = \langle x, \langle F(\overline{g})(x), \mathsf{refl}\rangle\rangle$, and also

$$\begin{aligned}\varphi(F(\overline{g})(x)) &= \langle F(\pi_0)(F(\overline{g})(x)), \langle F(\overline{g})(x), \mathsf{refl}\rangle\rangle \\ &= \langle F(\mathsf{id})(x), \langle F(\overline{g})(x), \mathsf{refl}\rangle\rangle \\ &= \langle x, \langle F(\overline{g})(x), \mathsf{refl}\rangle\rangle\end{aligned}$$

since $\pi_0 \circ \overline{g} = \mathsf{id}$. Hence $\overline{\overline{F}(g)} = \varphi \circ F(\overline{g})$. □

In fact, this property determines $\Box_F$ up to natural isomorphism:

Proposition 4.14 Let $X_F : (P : A \to \mathrm{Set}) \to F(A) \to \mathrm{Set}$. Then $X_F(P) \cong \Box_F(P)$ if and only if there is

$$\varphi : F(\Sigma\ A\ P) \xrightarrow{\cong} \Sigma\ (FA)\ (X_F P) \qquad (*_\Box)$$

such that

$$\pi_0 \circ \varphi = F(\pi_0)\ . \qquad (**_\Box)$$

Proof. ($\Rightarrow$) Let $\psi_x : \Box_F(P, x) \xrightarrow{\cong} X_F(P, x)$. From Lemma 4.13, we know that there is an isomorphism $\varphi_0 : F(\Sigma\ A\ P) \to \Sigma\ (FA)\ (\Box_F P)$ satisfying ($**_\Box$). Define $\varphi := [\mathsf{id}, \psi] \circ \varphi_0$. Then φ is an isomorphism (with inverse $\varphi_0^{-1} \circ [\mathsf{id}, \psi^{-1}]$) and the following diagram commutes:

$$\begin{tikzcd}
F(\Sigma\ A\ P) \arrow[r, "\varphi_0"] \arrow[dr, "F(\pi_0)"'] & \Sigma\ (F\ A)\ (\Box_F P) \arrow[r, "{[\mathsf{id},\psi]}"] \arrow[d, "\pi_0"] & \Sigma\ (F\ A)\ (XP) \arrow[dl, "\pi_0"] \\
& F\ A &
\end{tikzcd}$$

which shows that ($**_\Box$) holds.

($\Leftarrow$) Assume ($*_\Box$) and ($**_\Box$) holds. Then

$$\begin{aligned}X(P, x) &\cong \{z : \Sigma\ F(A)\ (X(P)) \mid \pi_0(z) = x\} \\ &\cong \{y : F(\Sigma\ A\ P) \mid \pi_0(\varphi(y)) = x\} \\ &= \{y : F(\Sigma\ A\ P) \mid F(\pi_0)(y) = x\} = \Box_F(P, x)\ .\end{aligned}$$ □

Thus, in the general framework, we will define $\Box_F$ to be any type having this property, after having done the necessary translation to the language of Categories with Families:

Definition 4.15 Let $F : \mathbb{C} \to \mathbb{D}$ be a functor between Categories with Families. We say that $\Box_F$ *exists,* if there for each object X in $\mathbb{C}$ and $P \in \mathrm{Ty}_{\mathbb{C}}(X)$ exists $\Box_F(X, P) \in \mathrm{Ty}_{\mathbb{D}}(F(X))$ such that there is an isomorphism

$$\varphi : F(X \cdot P) \to F(X) \cdot \Box_F(X, P)$$

with $\mathbf{p} \circ \varphi = F(\mathbf{p})$. ■

Remark 4.16 (CwF pseudo morphisms) Dybjer [1996] defines a morphism (F, σ) between Categories with Families $\mathbb{C}$ and $\mathbb{D}$ to be a functor $F : \mathbb{C} \to \mathbb{D}$ together with a natural transformation $\sigma : (\mathrm{Ty}_{\mathbb{C}}, (\vdash)_{\mathbb{C}}) \to (\mathrm{Ty}_{\mathbb{D}}, (\vdash)_{\mathbb{D}}) \circ F$ such that the terminal object and context comprehensions are preserved. Dybjer [1996] requires preservation on the nose, whereas Clairambault and Dybjer [2011] introduces *pseudo* CwF morphisms where the structure is preserved only up to isomorphism.

In a sense, Definition 4.15 is a generalisation of this concept: If (F, σ) is a pseudo CwF morphism, then $\Box_F$ exists, and is given by the first component of σ. Our perspective is slightly different: we consider $F : \mathbb{C} \to \mathbb{D}$ to be fixed (as the pattern functor for the data type we are interested in), and ask that there is a natural transformation $\Box_F : \mathrm{Ty}_{\mathbb{C}} \to \mathrm{Ty}_{\mathbb{D}} \circ F$ such that context comprehensions are preserved. Many functors F that we will consider will not preserve terminal objects, and we do not need them to. We will also see no need to require $\Box_F$ to act on $(\Gamma \vdash A)$.

If $\Box_F$ exists, then $\Box_F$ is unique up to isomorphism also in the general framework; for this to make sense, we need to make a category out of $\mathrm{Ty}(\Gamma)$. The following construction is due to Clairambault [2006, Section 4.1].

Definition 4.17 Let Γ be an object in the Category with Families $\mathbb{C}$. The category $\mathrm{Ty}(\Gamma)$ has as objects the elements from $\mathrm{Ty}(\Gamma)$, and

$$\mathsf{Hom}_{\mathrm{Ty}(\Gamma)}(A, B) = (\Gamma \cdot A \vdash B[\mathbf{p}]) \ .$$

Composition is given by $g \circ f \coloneqq g[\langle \mathbf{p}, f \rangle]$ (with identity $\mathbf{q}$). ◘

We can easily check that composition is associative

$$\begin{aligned}(g \circ f) \circ h &= g[\langle \mathbf{p}, f \rangle][\langle \mathbf{p}, h \rangle] \\ &= g[\langle \mathbf{p} \circ \langle \mathbf{p}, h \rangle, f[\langle \mathbf{p}, h \rangle] \rangle] \\ &= g[\langle \mathbf{p}, f[\langle \mathbf{p}, h \rangle] \rangle] = g \circ (f \circ h)\end{aligned}$$

and $\mathbf{q}$ really is an identity:

$$\mathbf{q} \circ f = \mathbf{q}[\langle \mathbf{p}, f \rangle] = f \qquad g \circ \mathbf{q} = g[\langle \mathbf{p}, \mathbf{q} \rangle] = g[\mathrm{id}] = g$$

Proposition 4.18 Suppose that $\Box_F(\Gamma, A)$ and $\Box'_F(\Gamma, A)$ with isomorphisms φ and φ' as in Definition 4.15 are given. Then they are isomorphic as objects in $\mathrm{Ty}(F(\Gamma))$.

Proof. Define

$$f \in (F(\Gamma) \cdot \Box'_F(\Gamma, A) \vdash \Box_F(\Gamma, A)[\mathbf{p}])$$

and

$$g \in (F(\Gamma) \cdot \Box_F(\Gamma, A) \vdash \Box'_F(\Gamma, A)[\mathbf{p}])$$

by $f = \mathbf{q}[\varphi \circ \varphi'^{-1}]$ and $g = \mathbf{q}[\varphi' \circ \varphi^{-1}]$. Both terms have the right type since $\mathbf{p} \circ \varphi \circ \varphi'^{-1} = F(\mathbf{p}) \circ \varphi'^{-1} = \mathbf{p}$ (and similarly for g). We calculate

$$\begin{aligned}f \circ g = f[\langle \mathbf{p}, g \rangle] &= \mathbf{q}[\varphi \circ \varphi'^{-1} \circ \langle \mathbf{p}, \mathbf{q}[\varphi' \circ \varphi^{-1}] \rangle] \\ &= \mathbf{q}[\varphi \circ \varphi'^{-1} \circ \langle \mathbf{p} \circ \varphi \circ \varphi'^{-1}, \mathbf{q}[\varphi' \circ \varphi^{-1}] \rangle] \qquad \text{since } \mathbf{p} = \mathbf{p} \circ \varphi \circ \varphi'^{-1} \\ &= \mathbf{q}[\varphi \circ \varphi'^{-1} \circ \langle \mathbf{p}, \mathbf{q} \rangle \circ \varphi' \circ \varphi^{-1}] \\ &= \mathbf{q}[\mathrm{id}] = \mathbf{q} = \mathsf{id}_{\mathrm{Ty}(F(\Gamma))}\end{aligned}$$

and similarly for $g \circ f$. Thus $\Box_F(\Gamma, A)$ and $\Box'_F(\Gamma, A)$ are isomorphic. □

We can give an alternative proof by noting that there is another characterisation of the morphisms of $\mathrm{Ty}_{\mathbb{C}}(\Gamma)$, which is due to Clairambault and Dybjer [2011]:

Lemma 4.19 $\mathrm{Ty}_{\mathbb{C}}(\Gamma)$ is isomorphic to the category $\mathrm{Ty}'_{\mathbb{C}}(\Gamma)$ with the same objects, but where the morphisms from A to B are morphisms $f : \Gamma \cdot A \to \Gamma \cdot B$ in $\mathbb{C}$ such that $\mathbf{p} \circ f = \mathbf{p}$.

Proof. Define $F : \mathrm{Ty}_{\mathbb{C}}(\Gamma) \to \mathrm{Ty}'_{\mathbb{C}}(\Gamma)$ and $G : \mathrm{Ty}'_{\mathbb{C}}(\Gamma) \to \mathrm{Ty}_{\mathbb{C}}(\Gamma)$ to be the identities on objects. For a morphisms $M \in (\Gamma \cdot A \vdash B[\mathbf{p}])$ and $f : \Gamma \cdot A \to \Gamma \cdot B$, let $F(M) = \langle \mathbf{p}, M \rangle$ and $G(f) = \mathbf{q}[f]$. Note that $\mathbf{p} \circ F(M) = \mathbf{p} \circ \langle \mathbf{p}, M \rangle = \mathbf{p}$. We calculate:

$$F(G(f)) = F(\mathbf{q}[f]) = \langle \mathbf{p}, \mathbf{q}[f] \rangle = \langle \mathbf{p} \circ f, \mathbf{q}[f] \rangle = \langle \mathbf{p}, \mathbf{q} \rangle \circ f = f$$

$$G(F(M)) = G(\langle \mathbf{p}, M \rangle) = v[\langle \mathbf{p}, M \rangle] = M$$ □

Thus, $\varphi \circ \varphi'^{-1} : F(\Gamma) \cdot \Box'_F(\Gamma, A) \to F(\Gamma) \cdot \Box_F(\Gamma, A)$ is obviously an isomorphism in $\mathrm{Ty}'_{\mathbb{C}}(F(\Gamma))$ (in the proof of Proposition 4.18, we proved that $\mathbf{p} \circ \varphi \circ \varphi'^{-1} = \mathbf{p}$). Hence $\Box_F(\Gamma, A)$ and $\Box'_F(\Gamma, A)$ are isomorphic also in $\mathrm{Ty}_{\mathbb{C}}(F(\Gamma))$.

We will soon see in Section 4.2.2.1 that $\Box_F$ often exists for general reasons. But first, let us define a generic elimination rule for a data type in the general Categories with Families framework, given that $\Box_F$ exists.

Definition 4.20 (Generic elimination rule) Let F, $G : \mathbb{C} \to \mathbb{D}$ be functors between Categories with Families such that $\Box_F$ and $\Box_G$ exists. Let (X, in) be an (F, G)-dialgebra. The generic elimination rule for (X, in) says that there is a term elim as follows:

$$\frac{P \in \mathrm{Ty}(X) \qquad \mathsf{step}_{\mathsf{in}} \in (F(X) \cdot \Box_F(P) \vdash \Box_G(P)[\mathsf{in} \circ \mathbf{p}])}{\mathsf{elim}(P, \mathsf{step}_{\mathsf{in}}) \in (X \vdash P)}$$ ∎

Note that (X, in) need not be an initial object in $\mathsf{Dialg}(F, G)$ for this definition to make sense – just as it makes sense to ask if e.g. $\mathbb{R}$ satisfies the induction principle for natural numbers (the answer is of course no). We will show in Section 4.3 that in fact the elimination rule is valid if and only if (X, in) is initial.

The reader might be puzzled by the fact that G and $\Box_G$ appear in the type of $\mathsf{step}_{\mathsf{in}}$, instead of e.g. (warning: this is wrong!)

$$\text{“}\mathsf{step}'_{\mathsf{in}} \in (F(X) \cdot \Box_F(P) \vdash P[\mathsf{in} \circ \mathbf{p}])\text{”}$$

which would be the Categories with Families representation of a dependent function (warning: still wrong!)

$$\text{“}\mathsf{step}'_{\mathsf{in}} : ((x, \tilde{x}) : \Sigma\, F(X)\ \Box_F\,(P)) \to P(\mathsf{in}(x))\text{”}\ .$$

Looking closer at the types, we see that this does not make sense. The predicate $P : \mathrm{Ty}_{\mathbb{C}}(X)$ lives in the Category with Families $\mathbb{C}$, but $\Box_F(P) : \mathrm{Ty}_{\mathbb{D}}(F(X))$ lives in the

Category with Families $\mathbb{D}$ – we cannot have a "function" whose domain and codomain live in different categories. We need a way to lift P from $\mathbb{C}$ to $\mathbb{D}$, but that is exactly what the predicate lifting $\Box_G$ does. Hence

$$\mathsf{step}_{\mathsf{in}} \in (F(X) \cdot \Box_F(P) \vdash \Box_G(P)[\mathsf{in} \circ \mathbf{p}])$$

i.e., in type-theoretical notation,

$$\mathsf{step}_{\mathsf{in}} : ((x, \tilde{x}) : \Sigma\, F(X)\ \ \Box_F\,(P)) \to \Box_G(P)(\mathsf{in}(x)) \ .$$

The type $\Box_G(P)$ can be seen as a lifting of P from types to predicates; in fact, this is what $\Box_G$ is usually called in the fibrational setting [Hermida and Jacobs, 1998]. In our applications, G will "morally" be the identity, which implies that $\Box_G$ will be so as well.

Lemma 4.21 If $G : \mathbb{C} \to \mathbb{D}$ is the identity on objects and G preserves context comprehensions and projections, i.e. $G(\Gamma \cdot_{\mathbb{C}} A) = \Gamma \cdot_{\mathbb{D}} A$ and $G(\mathbf{p}) = \mathbf{p}$, then $\Box_G$ exists, and $\Box_G(P) = P$ (with $\varphi = \mathsf{id}$).

Proof. Note that we have $\Box_G(P) \in \mathrm{Ty}(\Gamma) = \mathrm{Ty}(G(\Gamma))$. Since $G(\Gamma \cdot A) = \Gamma \cdot A$ and $G(\Gamma) = \Gamma$, the identity morphism is an isomorphism between $G(\Gamma \cdot A)$ and $G(\Gamma) \cdot A$. Furthermore, since $G(\mathbf{p}) = \mathbf{p}$, we trivially have $\mathbf{p} \circ \mathsf{id} = G(\mathbf{p})$. □

In particular, if we choose $G = \mathsf{Id} : \mathbb{C} \to \mathbb{C}$ and consider ordinary F-algebras for an endofunctor $F : \mathbb{C} \to \mathbb{C}$, the elimination rule becomes

$$\frac{P \in \mathrm{Ty}(X) \qquad \mathsf{step}_{\mathsf{in}} \in (F(X) \cdot \Box_F(P) \vdash P[\mathsf{in} \circ \mathbf{p}])}{\mathsf{elim}(P, \mathsf{step}_{\mathsf{in}}) \in (X \vdash P)}$$

4.2.2.1 Sufficient conditions for $\Box_F$ to exist

We now investigate conditions for $\Box_F$ to exist. As we will see, mild conditions on the Categories with Families involved will suffice. A Category with Families is said to support (extensional) identity types and Σ-types if it is closed under the following constructions respectively:

Definition 4.22 Let $\mathbb{C}$ be a Category with Families.

(i) $\mathbb{C}$ supports (extensional) identity types if

- $A \in \mathrm{Ty}(\Gamma)$ and $a, a' \in (\Gamma \vdash A)$ implies that there is $I_A(a, a') \in \mathrm{Ty}(\Gamma)$.
- $a \in (\Gamma \vdash A)$ implies that there is $r_{A,a} : (\Gamma, I_A(a \vdash a))$.
- $c : (\Gamma, I_A(a \vdash a'))$ implies that $a = a'$ and $c = r_{A,a}$.
- these constructions are stable under substitution, i.e.

$$\begin{aligned} I_A(a, a')[f] &= I_{A[f]}(a[f], a'[f]) \\ r_{A,a}[f] &= r_{A[f],a[f]} \end{aligned}$$

(ii) $\mathbb{C}$ supports Σ-types if

- $A \in \mathrm{Ty}(\Gamma)$ and $B \in \mathrm{Ty}(\Gamma \cdot A)$ implies that there is $\Sigma(A, B) \in \mathrm{Ty}(\Gamma)$.
- $a \in (\Gamma \vdash A)$ and $b \in (\Gamma \vdash B[\overline{a}])$ implies that there is $p(a, b) : (\Gamma \vdash \Sigma(A, B))$.
- $c : (\Gamma, \Sigma(A \vdash B))$ implies that there are $\pi_1(c) \in (\Gamma \vdash A)$ and $\pi_2(c) \in (\Gamma \vdash B[\overline{\pi_1(c)}])$ such that

$$\pi_1(p(a,b)) = a \qquad \pi_2(p(a,b)) = b \qquad p(\pi_1(c), \pi_2(c)) = c$$

- these constructions are stable under substitution, i.e.

$$\begin{aligned} \Sigma(A, B)[f] &= \Sigma(A[f], B[\langle f \circ \mathbf{p}, \mathbf{q} \rangle]) \\ p(a, b)[f] &= p(a[f], b[f]) \\ \pi_1(c)[f] &= \pi_1(c[f]) \\ \pi_2(c)[f] &= \pi_2(c[f]) \end{aligned}$$ ■

Both identity types and Σ-types are of course well-known from Type Theory. Hofmann [1997] shows how they can be interpreted in Categories with Families that support them. We are interested in slightly less well-known constructions, namely constant families and inverse image types.

Definition 4.23 A CwF $\mathbb{C}$ supports constant family types if the following data are given:

- For each Γ in $\mathbb{C}$, there is a type $\check{\Gamma}_\Delta \in \mathrm{Ty}(\Delta)$ for all Δ in $\mathbb{C}$ such that $\check{\Gamma}_\Delta[g] = \check{\Gamma}_B$ whenever $g : B \to \Delta$. (We will usually omit the subscript Δ.)
- There is an isomorphism $\cdot^\downarrow : (B \vdash \check{\Gamma}) \to \mathsf{Hom}(B, \Gamma)$ with inverse $\cdot^\uparrow : \mathsf{Hom}(B, \Gamma) \to (B \vdash \check{\Gamma})$ such that $M^\downarrow \circ g = M[g]^\downarrow$. ■

In the CwF Set (see Example 4.10), constant family types are simply constant families $\check{\Gamma}(x) = \Gamma$. The isomorphism $(B \vdash \check{\Gamma}) \cong \mathsf{Hom}(B, \Gamma)$ relates "non-dependant dependant" functions and ordinary (non-dependent) morphisms. Note that the equation $M^\downarrow \circ g = M[g]^\downarrow$ equivalently can be written

$$f \circ g^\uparrow = f^\uparrow[g]$$

by considering $f = M^\downarrow$ and applying $\cdot^\uparrow$ to both sides of the equation.

Clairambault and Dybjer [2011] defines a similar notion of *democracy* for a CwF; a CwF is democratic if each context is represented by a type. In detail:

Definition 4.24 (Clairambault and Dybjer [2011, Def. 6]) A CwF $\mathbb{C}$ is *democratic* if for each object Γ of $\mathbb{C}$ there is $\overline{\Gamma} \in \mathrm{Ty}(1_\mathbb{C})$ and an isomorphism $\gamma_\Gamma : \Gamma \to 1_\mathbb{C} \cdot \overline{\Gamma}$. ■

Reassuringly, constant families and democracy are interderivable.

Proposition 4.25 A CwF $\mathbb{C}$ supports constant families if and only if it is democratic.

Proof. ($\Rightarrow$) Assume $\mathbb{C}$ supports constant families. Define $\overline{\Gamma} := \check{\Gamma}_{\mathbf{1}_{\mathbb{C}}}$ and $\gamma_\Gamma = \langle !_\Gamma, \mathsf{id}^\uparrow \rangle : \Gamma \to \mathbf{1}_{\mathbb{C}} \cdot \overline{\Gamma}$ with inverse $\gamma_\Gamma^{-1} = \mathbf{q}^\downarrow : \mathbf{1}_{\mathbb{C}} \cdot \overline{\Gamma} \to \Gamma$. Of course, we have to check that γ_Γ and γ_Γ^{-1} are really inverse to each other:

$$\gamma_\Gamma^{-1} \circ \gamma_\Gamma = \mathbf{q}^\downarrow \circ \langle !_\Gamma, \mathsf{id}^\uparrow \rangle = (\mathbf{q}[\langle !_\Gamma, \mathsf{id}^\uparrow \rangle])^\downarrow = (\mathsf{id}^\uparrow)^\downarrow = \mathsf{id}$$

In the other direction, we have

$$\gamma_\Gamma \circ \gamma_\Gamma^{-1} = \langle !_\Gamma, \mathsf{id}^\uparrow \rangle \circ \mathbf{q}^\downarrow = \langle !_\Gamma \circ \mathbf{q}^\downarrow, \mathsf{id}^\uparrow[\mathbf{q}^\downarrow] \rangle = \langle !_{\mathbf{1}_{\mathbb{C}} \cdot \check{\Gamma}}, (\mathsf{id} \circ \mathbf{q}^\downarrow)^\uparrow \rangle = \langle \mathbf{p}, (\mathbf{q}^\downarrow)^\uparrow \rangle = \mathsf{id}$$

Here, we use that $\mathbf{p}(\check{\Gamma}) : \mathbf{1}_{\mathbb{C}} \cdot \check{\Gamma} \to \mathbf{1}_{\mathbb{C}}$ must be equal to $!_{\mathbf{1}_{\mathbb{C}} \cdot \check{\Gamma}} : \mathbf{1}_{\mathbb{C}} \cdot \check{\Gamma} \to \mathbf{1}_{\mathbb{C}}$ by the uniqueness of $!_{\mathbf{1}_{\mathbb{C}} \cdot \check{\Gamma}}$.

($\Leftarrow$) Assume $\mathbb{C}$ is democratic. Define $\check{\Gamma}_\Delta := \overline{\Gamma}[!_\Delta]$. Then

$$\check{\Gamma}_\Delta[g] = \overline{\Gamma}[!_\Delta \circ g] = \overline{\Gamma}[!_B] = \check{\Gamma}_B$$

by the uniqueness of $!_B$. Define $M^\downarrow := \gamma_\Gamma^{-1} \circ \langle !_B, M \rangle$ and $f^\uparrow := \mathbf{q}[\gamma_\Gamma \circ f]$. Then

$$(M^\downarrow)^\uparrow = \mathbf{q}[\gamma_\Gamma \circ \gamma_\Gamma^{-1} \circ \langle !_B, M \rangle] = \mathbf{q}[\langle !_B, M \rangle] = M$$

and using that $\mathbf{p} \circ \gamma_\Gamma \circ f : B \to \mathbf{1}_{\mathbb{C}}$ is equal to $!_B$ for every $f : B \to \Gamma$, we have

$$\begin{aligned}(f^\uparrow)^\downarrow &= \gamma_\Gamma^{-1} \circ \langle !_B, \mathbf{q}[\gamma_\Gamma \circ f] \rangle \\ &= \gamma_\Gamma^{-1} \circ \langle \mathbf{p} \circ \gamma_\Gamma \circ f, \mathbf{q}[\gamma_\Gamma \circ f] \rangle \\ &= \gamma_\Gamma^{-1} \circ \langle \mathbf{p}, \mathbf{q} \rangle \circ \gamma_\Gamma \circ f = f \quad .\end{aligned}$$

Finally, we have

$$\begin{aligned}M^\downarrow \circ g &= \gamma_\Gamma^{-1} \circ \langle !_B, M \rangle \circ g \\ &= \gamma_\Gamma^{-1} \circ \langle !_B \circ g, M[g] \rangle \\ &= \gamma_\Gamma^{-1} \circ \langle !_\Delta, M[g] \rangle \\ &= M[g]^\downarrow \quad .\end{aligned}$$

□

The second "type former" we need are inverse image types. These correspond to an indexed inductive definition in Type Theory; for $f : A \to B$, the inverse image type $\mathsf{InvIm}_f : B \to \mathsf{Set}$ is given by the constructor

$$\mathsf{im} : (x : A) \to \mathsf{InvIm}_f(f(x)) \quad ,$$

i.e. if $x : \mathsf{InvIm}_f(b)$ then $f(x) = b$. Alternatively, the inverse image of $f : A \to B$ can be defined as

$$\mathsf{InvIm}_f(b) \cong (\Sigma x : A)(f(x) \equiv_B b) \quad .$$

if we translate this to CwF combinators, we can define the inverse image f^* of f as

$$f^* := \Sigma(\breve{A}, I_{\breve{B}}(f^\uparrow[\mathbf{q}^\downarrow], \mathbf{p}^\uparrow)) \quad .$$

This makes sense whenever the CwF has Σ-types, identity types and constant families. We can define the "constructor" $\mathrm{im}_f : A \to B \cdot f^*$ by $\mathrm{im}_f := \langle f, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})\rangle$ since (checking that $r_{\breve{B},f^\uparrow}$ has the right type)

$$\begin{aligned}(A \vdash I_{\breve{B}}(f^\uparrow[\mathbf{q}^\downarrow], \mathbf{p}^\uparrow)[\langle f \circ \mathbf{p}, \mathbf{q}\rangle][\langle \mathsf{id}, \mathsf{id}^\uparrow\rangle]) &= (A \vdash I_{\breve{B}}(f^\uparrow[\mathbf{q}^\downarrow], \mathbf{p}^\uparrow)[\langle f, \mathsf{id}^\uparrow\rangle]) \\ &= (A \vdash I_{\breve{B}}(f^\uparrow[\mathbf{q}^\downarrow \circ \langle f, \mathsf{id}^\uparrow\rangle], \mathbf{p}^\uparrow[\langle f, \mathsf{id}^\uparrow\rangle])) \\ &= (A \vdash I_{\breve{B}}(f^\uparrow[(\mathsf{id}^\uparrow)^\downarrow], \mathbf{p}^\uparrow[\langle f, \mathsf{id}^\uparrow\rangle])) \\ &= (A \vdash I_{\breve{B}}(f^\uparrow, f^\uparrow)) \ .\end{aligned}$$

We are interested in inverse image types, since they give us a way to construct $\Box_F$. Let us first prove a preliminary lemma:

Lemma 4.26 (Clairambault and Dybjer [2011, Lemma 25]) Let $\mathbb{C}$ be a CwF with inverse image types. For all $f : A \to B$ in $\mathbb{C}$, we have an isomorphism $\varphi : B \cdot f^* \to A$ in $\mathbb{C}$ such that the following diagram commutes:

$$\begin{tikzcd}
B \cdot f^* \arrow[rr, bend left, "\varphi"] \arrow[dr, "\mathbf{p}"'] & & A \arrow[ll, bend left, "\varphi^{-1}"] \arrow[dl, "f"] \\
& B &
\end{tikzcd}$$

Proof. The isomorphism φ can be defined as $\varphi := \pi_1(\mathbf{q})^\downarrow$ with inverse $\varphi^{-1} := \mathrm{im}_f = \langle f, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})\rangle$. We then immediately have

$$\varphi \circ \varphi^{-1} = \pi_1(\mathbf{q})^\downarrow \circ \langle f, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})\rangle = \pi_1(\mathbf{q}[\langle f, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})\rangle])^\downarrow = (\mathsf{id}^\uparrow)^\downarrow = \mathsf{id}$$

In the other direction, we have

$$\begin{aligned}\varphi^{-1} \circ \varphi &= \langle f, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})\rangle \circ \pi_1(\mathbf{q})^\downarrow \\ &= \langle f \circ \pi_1(\mathbf{q})^\downarrow, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})[\pi_1(\mathbf{q})^\downarrow]\rangle \\ &= \langle f \circ \pi_1(\mathbf{q})^\downarrow, p(\pi_1(\mathbf{q}), r)\rangle\end{aligned}$$

If we can prove $f \circ \pi_1(\mathbf{q})^\downarrow = \mathbf{p}$ and $r = \pi_2(\mathbf{q})$, we are done, since then

$$\varphi^{-1} \circ \varphi = \langle \mathbf{p}, p(\pi_1(\mathbf{q}), \pi_2(\mathbf{q}))\rangle = \langle \mathbf{p}, \mathbf{q}\rangle = \mathsf{id}$$

by surjective pairing for Σ-types. But $\pi_2(\mathbf{q}) \in (B \cdot f^* \vdash I_{\breve{B}}(f^\uparrow[\mathbf{q}^\downarrow], \mathbf{p}^\uparrow)[\langle \mathbf{p}, \pi_1(\mathbf{q})\rangle])$, hence by the extensionality of identity types

$$f^\uparrow[\pi_1(\mathbf{q})^\downarrow] = \mathbf{p}^\uparrow$$

or equivalently $f \circ \pi_1(\mathbf{q})^\downarrow = \mathbf{p}$. Furthermore, indeed $\pi_2(\mathbf{q}) = r$ by the uniqueness of identity proofs.

Finally, $\mathbf{p} \circ \varphi^{-1} = \mathbf{p} \circ \langle f, p(\mathsf{id}^\uparrow, r_{\breve{B},f^\uparrow})\rangle = f$, so the diagram commutes. □

Proposition 4.27 Let $F : \mathbb{C} \to \mathbb{D}$ with $\mathbb{D}$ a CwF with inverse image types. Then $\Box_F(\Gamma, \sigma) \cong F(\mathbf{p})^*$.

Proof. By Lemma 4.26, $F(\mathbf{p})^*$ satisfies the universal property of $\Box_F(\Gamma, \sigma)$. □

By reformulating a theorem due to Hofmann [1994], who in turn adapted a construction due to Bénabou [1985], Clairambault and Dybjer [2011] proves:

Theorem 4.28 (Clairambault and Dybjer [2011, Lemma 18]) Let $\mathbb{C}$ be a category with finite limits. Then $\mathbb{C}$ can be extended to a CwF with constant families, extensional identity types and Σ-types. □

Hence, in order to see if a category $\mathbb{C}$ is a Category with Families or not, we only need to check if it has finite limits. For example, we immediately see that all Set^I for a fixed set I is are CwFs, since limits in functor categories are calculated pointwise. Set^I is the Category with Families which correspond to indexed inductive definitions.

Putting together the results from this section, we get:

Corollary 4.29 Let $\mathbb{D}$ be a category with finite limits, $\mathbb{C}$ a CwF. Then $\Box_F$ exists for any $F : \mathbb{C} \to \mathbb{D}$. □

4.2.3 Generic computation rules

So far, we have treated the induction hypothesis involved in the elimination rules. This is enough for a static view of data types and induction, but not for us: we want our proofs to compute. As briefly touched on at the beginning of Section 4.2.2, in the case of ordinary inductive types, modelled as initial algebras of endofunctors on Set, we expect a computation rule of the form

$$\mathsf{elim}_F(P, \mathsf{step}_c, c(x)) = \mathsf{step}_c(x, \overline{F}(\mathsf{elim}_F(P, \mathsf{step}_c), x)) \ .$$

where $\overline{F} : (f : (x : X) \to P(x)) \to (x : F(X)) \to \Box_F(P, x)$ is a kind of "dependent map function" that takes care of recursive calls by applying its input f in a way compatible with the structure dictated by F. The function $\overline{F}$ is like the action of F on morphisms, were it not for the fact that the input $f : (x : X) \to P(x)$ and output $\overline{F}(f) : (x : F(X)) \to \Box_F(P, x)$ are dependent functions, hence not morphisms in Set. However, if we make f non-dependent by considering $\overline{f} = \lambda x.\, \langle x, f(x) \rangle : X \to \Sigma\, X\, P$, then use $F(\overline{f})$ and the isomorphism $\varphi : F(\Sigma\, X\, P) \to \Sigma\, F(X)\, \Box_F(P)$, and take the second component, we end up with a dependent function of the right type. Since $\pi_0 \circ \varphi = F(\pi_0)$ by Lemma 4.13, $\pi_0 \circ \varphi \circ F(\overline{f}) = F(\pi_0 \circ \overline{f}) = F(\mathsf{id}) = \mathsf{id}$, and we can define

$$\overline{F}(f) \coloneqq \pi_1 \circ \varphi \circ F(\overline{f}) : (x : F(A)) \to \Box_F(P, (\pi_0 \circ \varphi \circ F(\overline{f}))(x)) \ . \qquad (4.3)$$

Example 4.30 (Computation rules for the type of natural numbers) Recall from Example 4.12 that the induction hypothesis type $\Box_{\lambda X.\, 1+X}$ for the natural numbers satisfy

$$\Box_{\lambda X.\, 1+X}(P, \mathsf{inl}(\star)) = \mathbf{1} \qquad \Box_{\lambda X.\, 1+X}(P, \mathsf{inr}(n)) = P(n) \ .$$

By the definition above, $\overline{\lambda X.\mathbf{1}+X}(P) : ((x : Y) \to P(x)) \to (x : \mathbf{1}+Y) \to \Box_{\lambda X.\mathbf{1}+X}(P, x)$ satisfies

$$\begin{aligned}\overline{\lambda X.\mathbf{1}+X}(f, \mathsf{inl}(\star)) &= \star \\ \overline{\lambda X.\mathbf{1}+X}(f, \mathsf{inr}(n)) &= f(n)\end{aligned}$$

Indeed, we then end up with the usual computation rules, after we have decomposed $\mathsf{step}_{[0,\mathsf{suc}]} : (x : \mathbf{1} + \mathbb{N}) \to \Box_{\lambda X.\mathbf{1}+X}(P, x) \to P(x)$ into $\mathsf{step}_{[0,\mathsf{suc}]} = [\mathsf{step}_0, \mathsf{step}_{\mathsf{suc}}]$ where $\mathsf{step}_0 : \mathbf{1} \to P(0)$ and $\mathsf{step}_{\mathsf{suc}} : (n : \mathbb{N}) \to P(n) \to P(\mathsf{suc}(n))$:

$$\begin{aligned}\mathsf{elim}_{\lambda X.\mathbf{1}+X}(P, [\mathsf{step}_0, \mathsf{step}_{\mathsf{suc}}], 0) &= \mathsf{step}_0(\star) \\ \mathsf{elim}_{\lambda X.\mathbf{1}+X}(P, [\mathsf{step}_0, \mathsf{step}_{\mathsf{suc}}], \mathsf{suc}(n)) &= \mathsf{step}_{\mathsf{suc}}(n, \mathsf{elim}_{\lambda X.\mathbf{1}+X}(P, [\mathsf{step}_0, \mathsf{step}_{\mathsf{suc}}], n)) \quad \blacksquare\end{aligned}$$

We now generalise $\overline{F}(P) : (f : (x : X) \to P(x)) \to (x : F(X)) \to \Box_F(P, x)$ to the generic setting, by replacing predicates with types and dependent function spaces with terms from the category with families.

Definition 4.31 Let $F : \mathbb{C} \to \mathbb{D}$ be a functor between Categories with Families such that $\Box_F$ exists. For each object X in $\mathbb{C}$ and $P \in \mathrm{Ty}(X)$ we define

$$\overline{F} : (X \vdash P) \to (F(X) \vdash \Box_F(P))$$

by

$$\overline{F}(f) = \mathbf{q}[\varphi_F \circ F(\overline{f})] \qquad \blacksquare$$

We see that $\overline{F}$ indeed coincides with the definition in (4.3) for the CwF Set.

Lemma 4.32 Let $F : \mathbb{C} \to \mathbb{D}$ be a functor between Categories with Families such that $\Box_F$ exists.

(i) $\overline{\mathsf{Id}} = \mathsf{Id}$.

(ii) $\overline{\overline{F}(f)} = \varphi_F \circ F(\overline{f})$ for all $f \in (X \vdash P)$.

Proof.

(i) $\overline{\mathsf{Id}}(f) = \mathbf{q}[\varphi_{\mathsf{Id}} \circ \mathsf{Id}(\overline{f})] = \mathbf{q}[\overline{f}] = f$.

(ii) We calculate

$$\begin{aligned}\overline{\overline{F}(f)} &= \overline{\mathbf{q}[\varphi_F \circ F(\overline{f})]} \\ &= \langle \mathsf{id}, \mathbf{q}[\varphi_F \circ F(\overline{f})] \rangle \\ &= \langle F(\mathbf{p} \circ \overline{f}), \mathbf{q}[\varphi_F \circ F(\overline{f})] \rangle \\ &= \langle \mathbf{p} \circ \varphi_F \circ F(\overline{f}), \mathbf{q}[\varphi_F \circ F(\overline{f})] \rangle \\ &= \varphi_F \circ F(\overline{f}) \qquad \Box\end{aligned}$$

We now state the generic computation rule:

Definition 4.33 Let $F, G : \mathbb{C} \to \mathbb{D}$ be functors between Categories with Families such that $\Box_F$ and $\Box_G$ exist. Let (X, in) be an (F, G)-dialgebra. The *computation rule* associated with an elimination rule

$$\frac{P \in \mathrm{Ty}(X) \qquad \mathsf{step_{in}} \in (F(X) \cdot \Box_F(P) \vdash \Box_G(P)[\mathsf{in} \circ \mathbf{p}])}{\mathsf{elim}(P, \mathsf{step_{in}}) \in (X \vdash P)}$$

says that

$$\overline{G}(\mathsf{elim}(P, \mathsf{step_{in}}))[\mathsf{in}] = \mathsf{step_{in}}[\overline{\overline{F}(\mathsf{elim}(P, \mathsf{step_{in}}))}] \qquad \blacksquare$$

Notice the similarity with a morphism $h : (X, \mathsf{in}) \to (P, \mathsf{step})$ in $\mathsf{Dialg}(F, G)$, which is a morphism $h : X \to P$ such that $G(h) \circ \mathsf{in} = \mathsf{step} \circ F(h)$.[4]

Example 4.34 (The elimination rule for F-algebras on Set) Let $F : \mathsf{Set} \to \mathsf{Set}$ be an endofunctor and $\mathsf{Id} : \mathsf{Set} \to \mathsf{Set}$ the identity functor. For (F, Id)-dialgebras (i.e. ordinary F-algebras) the elimination rule becomes (after currying $\mathsf{step_{in}}$)

$$\frac{P : X \to \mathsf{Set} \qquad \mathsf{step_{in}} : (x : F(X)) \to \Box_F(P, x) \to P(\mathsf{in}(x))}{\mathsf{elim}(P, \mathsf{step_{in}}) : (x : X) \to P(x)}$$

as we are used to. The computation rule becomes

$$\mathsf{elim}(P, \mathsf{step_{in}}, \mathsf{in}(x)) = \mathsf{step_{in}}(x, \overline{F}(P, \mathsf{elim}(P, \mathsf{step_{in}}), x))$$

for $x : F(X)$. $\blacksquare$

4.2.4 The generic eliminator for an inductive-inductive definition

Recall from Section 4.1.1 that inductive-inductive definitions are represented by functors $\mathsf{Arg} : \mathsf{Dialg}(\mathsf{Arg_A}, U) \to \mathsf{Fam}(\mathsf{Set})$. We want to show that $\Box_{\mathsf{Arg}}$ exists. In order to apply Corollary 4.29, we need to check that $\mathsf{Dialg}(\mathsf{Arg_A}, U)$ is a Category with Families and that $\mathsf{Fam}(\mathsf{Set})$ has finite limits. The second requirement is easily taken care of since Set is complete:

Proposition 4.35 If $\mathbb{C}$ has finite limits, then so does $\mathsf{Fam}\,\mathbb{C}$ and the index set functor $U : \mathsf{Fam}\,\mathbb{C} \to \mathsf{Set}$ preserves them.

Proof. It is enough for $\mathsf{Fam}\,\mathbb{C}$ to have a terminal object and pullbacks [Mac Lane, 1998]. The terminal object in $\mathsf{Fam}\,\mathbb{C}$ is $(\mathbf{1}, \lambda\_.\, \mathbf{1}_{\mathbb{C}})$. We construct the pullback of $(f, g) : (A, B) \to (C, D)$ and $(f', g') : (A', B') \to (C, D)$ as in

$$\begin{tikzcd}
(A, B) \times_{(C,D)} (A', B') \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & (A', B') \arrow[d, "(f', g')"] \\
(A, B) \arrow[r, "(f, g)"'] & (C, D)
\end{tikzcd}$$

[4] We have used the unorthodox variable names (P, step) for a dialgebra here to show the similarity with the situation above.

by $(A,B) \times_{(C,D)} (A',B') := (A \times_C A', \lambda\langle x,y\rangle. B(x) \times_{D(f(x))} B'(y))$ where $A \times_C A' = \{\langle x,y\rangle : A \times A' \mid f(x) = f'(y)\}$ is the usual construction of the pullback of f and f' in Set, and $B(x) \times_{D(f(x))} B'(y)$ is the pullback of $g_x : B(x) \to D(f(x))$ and $g'_y : B'(y) \to D(f'(y))$ (with common codomain, since $f(x) = f'(y)$) in $\mathbb{C}$. The projections are constructed from the projections in Set and $\mathbb{C}$. □

In particular, by Theorem 4.28, this shows that $\mathsf{Fam}(\mathsf{Set})$ can be extended to a Category with Families. For later reference, we can give a Categories with Families structure for $\mathsf{Fam}(\mathsf{Set})$ explicitly:

Example 4.36 ($\mathsf{Fam}(\mathsf{Set})$ as a CwF) The category $\mathsf{Fam}(\mathsf{Set})$ can be made into a Category with Families if we define

$$\begin{aligned}
\mathrm{Ty}(X,Y) &= \{(A,B) \mid A : X \to \mathsf{Set}, B : (x : X) \to Y(x) \to A(x) \to \mathsf{Set}\}\\
((X,Y) \vdash (A,B)) &= \{(h,k) \mid h : \prod_{x \in X} A(x), k : \prod_{x \in X, y \in Y(x)} B(x,y,h(x))\}
\end{aligned}$$

For $(f,g) : (X,Y) \to (X',Y')$, we define

$$\begin{aligned}
&(A,B)[f,g] : \mathrm{Ty}(X,Y) = \{(A,B) \mid A : X \to \mathsf{Set}, B : (x : X) \to Y(x) \to A(x) \to \mathsf{Set}\}\\
&(A,B)[f,g] = (A,B) \circ (f,g) = (A \circ f, \lambda x.\lambda y. B(f(x), g(x,y))\\
&\quad (h,k)[f,g] : ((X,Y) \vdash (A,B)[f,g])\\
&\quad (h,k)[f,g] = (h,k) \circ (f,g) = (h \circ f, \lambda x.\lambda y. k(f(x), g(x,y)))
\end{aligned}$$

The context comprehension can be given by

$$\begin{aligned}
(X,Y) \cdot (A,B) &= (\sum_{x \in X} A(x), \lambda\langle x,a\rangle. \sum_{y \in Y(x)} B(x,y,a))\\
\mathrm{p}(A,B) &= (\mathsf{fst}, \lambda x. \mathsf{fst})\\
\mathrm{q}_{A,B} &= (\mathsf{snd}, \lambda x. \mathsf{snd})
\end{aligned}$$

Given $(f,g) : (X',Y') \to (X,Y)$ and $(h,k) \in ((X',Y') \vdash (A,B)[f,g])$, we have

$$(\theta,\psi) = \langle (f,g),(h,k)\rangle_{(A,B)} : (X',Y') \to (X,Y) \cdot (A,B)$$

defined as $\theta(x) = \langle f(x), h(x)\rangle$ and $\psi(x,y) = \langle g(x,y), k(x,y)\rangle$. ■

We now show that $\mathsf{Dialg}(\mathsf{Arg}_\mathsf{A}, U)$ is a category with families. In fact, we show that if $\mathbb{C}$ has finite limits and $G : \mathbb{C} \to \mathbb{D}$ preserves them, then also $\mathsf{Dialg}(F,G)$ has finite limits, and hence is a Category with Families by Theorem 4.28. The following is a straightforward generalisation of the well-known corresponding folklore theorem for the category of F-algebras, i.e. the case of $G = \mathsf{Id} : \mathbb{C} \to \mathbb{C}$.

Theorem 4.37 Let $F,G : \mathbb{C} \to \mathbb{D}$. The category $\mathsf{Dialg}(F,G)$ has finite limits if $\mathbb{C}$ does, and G preserves them. The forgetful functor $V : \mathsf{Dialg}(F,G) \to \mathbb{C}$ preserves finite limits.

Proof. Define $\mathbf{1}_{\mathsf{Dialg}(F,G)} := (\mathbf{1}_{\mathbb{C}}, !_{F(\mathbf{1}_{\mathbb{C}})})$ where $!_{F(\mathbf{1}_{\mathbb{C}})}$ is the unique map $F(\mathbf{1}_{\mathbb{C}}) \to \mathbf{1}_D$. For any object (X, f), the unique morphism $(X, f) \to (\mathbf{1}_{\mathbb{C}}, !_{F(\mathbf{1}_{\mathbb{C}})})$ is given by the unique arrow $!_X$ from X to $\mathbf{1}_C$ in $\mathbb{C}$, and the diagram

$$\begin{tikzcd}
FX \arrow[r, "f"] \arrow[d, "F(!_X)"'] & GX \arrow[d, "G(!_X)"] \\
F(\mathbf{1}_C) \arrow[r, "!_{F(\mathbf{1}_C)}"'] & G(\mathbf{1}_{\mathbb{C}}) = \mathbf{1}_{\mathbb{D}}
\end{tikzcd}$$

commutes since both paths are arrows into $\mathbf{1}_{\mathbb{D}}$, hence equal.

We now construct the pullback of $f : (A, a) \to (C, c)$ and $g : (B, b) \to (C, c)$. Let $\psi : G(A) \times_{G(C)} G(B) \to G(A \times_C B)$ be the isomorphism that witnesses that G preserves pullbacks. The carrier and the projections of the pullback of f and g are inherited from $\mathbb{C}$. In detail, we construct the pullback as $(A \times_C B, \psi \circ \langle a \circ F(p), b \circ F(q) \rangle)$ where $A \times_C B$ is the pullback of f and g in $\mathbb{C}$, with projections $p : A \times_C B \to A$ and $q : A \times_C B \to B$, and $\langle -, - \rangle$ is the mediating morphism given by the universal property of $G(A) \times_{G(C)} G(B)$. An easy diagram chase shows that all morphisms involved also are morphisms in $\mathsf{Dialg}(F, G)$:

$$\begin{tikzcd}
& F(A \times_C B) \arrow[rr, "F(q)"] \arrow[dl, "\psi\circ\langle a\circ F(p), b\circ F(q)\rangle"'] \arrow[dd, "F(p)"] & & F(B) \arrow[dl, "b"] \arrow[dd, "F(g)"] \\
G(A \times_C B) \arrow[rr, "G(q)"'] \arrow[dd, "G(p)"'] & & G(B) \arrow[dd, "G(g)"] & \\
& F(A) \arrow[rr, dashed, "F(f)"] \arrow[dl, "a"'] & & F(C) \arrow[dl, "c"] \\
G(A) \arrow[rr, "G(f)"'] & & G(C) &
\end{tikzcd}$$

□

Since $\mathsf{Fam}(\mathsf{Set})$ has finite limits and the index set functor $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ preserves them by Proposition 4.35, we can apply Theorem 4.37 in conjunction with Theorem 4.28 to conclude that $\mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U)$ is a Category with Families.

Furthermore, by Corollary 4.29, the induction hypothesis type $\Box_{\mathsf{Arg}}$ exists for all $\mathsf{Arg} : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$. Explicitly, it can be defined as follows.

Example 4.38 ($\Box_{\mathsf{Arg}}$ exist) We can decompose $\Box_{\mathsf{Arg}} = (\Box_{\mathsf{Arg}_{\mathsf{A}}}, \Box_{\mathsf{Arg}_{\mathsf{B}}})$ into two components with the following types:

$$\begin{aligned}
&\Box_{\mathsf{Arg}_{\mathsf{A}}}(P, Q) : \mathsf{Arg}_{\mathsf{A}}(A, B) \to \mathsf{Set} \quad , \\
&\Box_{\mathsf{Arg}_{\mathsf{B}}}(P, Q) : \big(\mathsf{step}_c : (x : \mathsf{Arg}_{\mathsf{A}}(A, B)) \to \Box_{\mathsf{Arg}_{\mathsf{A}}}(P, Q, x) \to P(c(x))\big) \to \\
&\qquad (x : \mathsf{Arg}_{\mathsf{A}}(A, B)) \to (y : \mathsf{Arg}_{\mathsf{B}}(A, B, c, x)) \to \\
&\qquad (\widetilde{x} : \Box_{\mathsf{Arg}_{\mathsf{A}}}(P, Q, x)) \to \mathsf{Set}
\end{aligned}$$

with the following definitions

$$\begin{aligned}\Box_{\mathsf{Arg_A}}(P,Q,x) &:= \{y : \mathsf{Arg_A}((A,B)\cdot_{\mathsf{Fam(Set)}}(P,Q))\ |\mathsf{Arg_A}(\pi_0,\pi_0')(y) = x\}\ ,\\ \Box_{\mathsf{Arg_B}}(P,Q,\mathsf{step}_c,x,y,\widetilde{x}) &:= \\ &\{z : \mathsf{Arg_B}((\Sigma_{\mathsf{Dialg}}(A,B,c)\ (P,Q,\mathsf{step}_c)),\widetilde{x})\ |\ \mathsf{Arg_B}(\pi_0,\pi_0',\widetilde{x},z) = y\}\ ,\end{aligned}$$

Here,

$$(A,B)\cdot_{\mathsf{Fam(Set)}}(P,Q) = (\Sigma\ A\ P, \lambda\langle a,p\rangle.\,\Sigma b{:}B(a).\ Q(a,b,p))$$

and

$$(A,B,c)\cdot_{\mathsf{Dialg(Arg_A},U)}(P,Q,\mathsf{step}_c) = ((A,B)\cdot_{\mathsf{Fam(Set)}}(P,Q),[c,\mathsf{step}_c]\circ\varphi_{\mathsf{Arg_A}})$$

■

We know that $\mathsf{Dialg}(\mathsf{Arg},V)$ is a Category with Families by Theorem 4.37. However, it is easy to see that $\mathbb{E}_{\mathsf{Arg}}$ as a subcategory is closed under context comprehension $-\cdot-$ and substitution $-[-]$, i.e. if $\Gamma : \mathbb{E}_{\mathsf{Arg}}$ and $\sigma : \mathrm{Ty}(E_{\mathsf{Arg}}(\Gamma))$, where $E_{\mathsf{Arg}} : \mathbb{E}_{\mathsf{Arg}} \hookrightarrow \mathsf{Dialg}(\mathsf{Arg},V)$ is the embedding given by the equaliser, then also $\Gamma\cdot\sigma$ is in $\mathbb{E}_{\mathsf{Arg}}$, and similarly for substitution. Hence, $\mathbb{E}_{\mathsf{Arg}}$ inherits a category with families structure from $\mathsf{Dialg}(\mathsf{Arg},V)$:

Corollary 4.39 The category $\mathbb{E}_{\mathsf{Arg}}$ is a Category with Families. □

In general, an eliminator for (A,B,c,d) in $\mathbb{E}_{\mathsf{Arg}}$ is a term of the form

$$\begin{array}{c} P : A \to \mathsf{Set}\\ Q : (x:A) \to B(x) \to P(x) \to \mathsf{Set}\\ \mathsf{step}_c : (x:\mathsf{Arg_A}(A,B)) \to \Box_{\mathsf{Arg_A}}(P,Q,x) \to P(c(x))\\ \mathsf{step}_d : (x:\mathsf{Arg_A}(A,B)) \to (y:\mathsf{Arg_B}(A,B,c,x)) \to (\widetilde{x} : \Box_{\mathsf{Arg_A}}(P,Q,x))\\ \qquad\qquad \to \Box_{\mathsf{Arg_B}}(P,Q,c,\mathsf{step}_c,x,y,\widetilde{x}) \to Q(c(x),d(x,y),\mathsf{step}_c(x,\widetilde{x}))\\ \hline \mathsf{elim}_{\mathsf{Arg_A}}(P,Q,\mathsf{step}_c,\mathsf{step}_d) : (x:A) \to P(x)\\ \mathsf{elim}_{\mathsf{Arg_B}}(P,Q,\mathsf{step}_c,\mathsf{step}_d) : (x:A) \to (y:B(x)) \to Q(x,y,\mathsf{elim}_{\mathsf{Arg_A}}(P,Q,\mathsf{step}_c,\mathsf{step}_d,x)) \end{array}$$

with

$$\begin{aligned}\mathsf{elim}_{\mathsf{Arg_A}}(P,Q,\mathsf{step}_c,\mathsf{step}_d,c(x)) &= \mathsf{step}_c(x,\overline{\mathsf{Arg_A}}')\\ \mathsf{elim}_{\mathsf{Arg_B}}(P,Q,\mathsf{step}_c,\mathsf{step}_d,c(x),d(x,y)) &= \mathsf{step}_d(x,y,\overline{\mathsf{Arg_A}}',\overline{\mathsf{Arg_B}}')\end{aligned}$$

where

$$\begin{aligned}\overline{\mathsf{Arg_A}}' &= \overline{\mathsf{Arg_A}}(\mathsf{elim}_{\mathsf{Arg_A}}(P,Q,\mathsf{step}_c,\mathsf{step}_d),\mathsf{elim}_{\mathsf{Arg_B}}(P,Q,\mathsf{step}_c,\mathsf{step}_d),x)\\ \overline{\mathsf{Arg_B}}' &= \overline{\mathsf{Arg_B}}(\mathsf{step}_c,\mathsf{elim}_{\mathsf{Arg_A}}(P,Q,\mathsf{step}_c,\mathsf{step}_d),\mathsf{elim}_{\mathsf{Arg_B}}(P,Q,\mathsf{step}_c,\mathsf{step}_d),x,y)\ .\end{aligned}$$

Example 4.40 (The eliminator for sorted lists) Recall from Example 4.5 that sorted lists were given by the functors $\mathsf{Arg}_{\mathsf{SList}}$, $\mathsf{Arg}_{\leq_{\mathrm{L}}}$, where

$$\mathsf{Arg}_{\mathsf{SList}}(A, B) = \mathbf{1} + (\Sigma\, n{:}\mathbb{N}.\, \Sigma\, \ell{:}A.\, B(n, \ell))$$

Thus, we see that e.g.

$$\begin{aligned}
\Box_{\mathsf{Arg}_{\mathsf{SList}}}(P, Q, \mathsf{inl}(\star)) &= \{y : \mathbf{1} + \ldots \mid (\mathsf{id} + \ldots)(y) = \mathsf{inl}(\star)\} \cong \mathbf{1} \\
\Box_{\mathsf{Arg}_{\mathsf{SList}}}(P, Q, \mathsf{inr}(\langle n, \ell, p\rangle)) &\cong \\
\{y : \Sigma\, n'{:}\mathbb{N}.\, \Sigma\, \langle \ell', \widetilde{\ell}\rangle{:}(\Sigma A P).\, \Sigma p'{:}B(n,\ell).\, Q(n', \ell', p', \widetilde{\ell}) &\mid \Sigma(\mathsf{id}, \Sigma(\pi_0, \pi_0'))(y) = \langle n, \ell, p\rangle\} \\
&\cong \Sigma \widetilde{\ell}{:}P(\ell).\, Q(n, \ell, p, \widetilde{\ell})
\end{aligned}$$

and similarly for $\Box_{\mathsf{Arg}_{\leq_{\mathrm{L}}}}$, so that the eliminators are equivalent to

$$\begin{aligned}
\mathsf{elim}_{\mathsf{SortedList}} : {} & (P : \mathsf{SortedList} \to \mathsf{Set}) \to \\
& (Q : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to n \leq_{\mathrm{L}} \ell \to P(\ell) \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\mathsf{nil}} : P(\mathsf{nil})) \to \\
& \big(\mathsf{step}_{\mathsf{cons}} : (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \to (\widetilde{\ell} : P(\ell)) \\
& \qquad \to Q(n, \ell, p, \widetilde{\ell}) \to P(\mathsf{cons}(n, \ell, p))\big) \to \\
& \big(\mathsf{step}_{\mathsf{triv}} : (n : \mathbb{N}) \to Q(n, \mathsf{nil}, \mathsf{triv}_n, \mathsf{step}_{\mathsf{nil}})\big) \to \\
& \big(\mathsf{step}_{\ll\cdot\gg} : (m : \mathbb{N}) \to (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \\
& \qquad \to (q : m \leq n) \to (p' : m \leq_{\mathrm{L}} \ell) \to (\widetilde{\ell} : P(\ell)) \\
& \qquad \to (\widetilde{p} : Q(n, \ell, p, \widetilde{\ell})) \to (\widetilde{p'} : Q(m, \ell, p', \widetilde{\ell})) \\
& \qquad \to Q(m, \mathsf{cons}(n, \ell, p), \ll q, p' \gg, \mathsf{step}_{\mathsf{cons}}(n, \ell, p, \widetilde{\ell}, \widetilde{p}))\big) \to \\
& (\ell : \mathsf{SortedList}) \to P(\ell) \quad , \\
\mathsf{elim}_{\leq_{\mathrm{L}}} : {} & \ldots \to \\
& (n : \mathbb{N}) \to (\ell : \mathsf{SortedList}) \to (p : n \leq_{\mathrm{L}} \ell) \\
& \qquad \to Q(n, \ell, p, \mathsf{elim}_{\mathsf{SortedList}}(P, Q, \mathsf{step}_{\mathsf{nil}}, \mathsf{step}_{\mathsf{cons}}, \mathsf{step}_{\mathsf{triv}}, \mathsf{step}_{\ll\cdot\gg}, \ell)) \quad .
\end{aligned}$$

■

4.3 The equivalence between having an eliminator and being initial

We now show that a dialgebra is an initial object if and only if it has an eliminator. Thus, in particular, we give a categorical characterisation of inductive-inductive definitions by instantiating the framework in an appropriate way. The result also applies to many other concrete classes of data types such as e.g. indexed inductive definitions. By working in this abstract setting, the proofs become more transparent as only the relevant details remain. We emphasise again that we are working in extensional Type Theory in this chapter.

4.3.1 Initiality implies the elimination rules

Theorem 4.41 Let $F, G : \mathbb{C} \to \mathbb{D}$ with $\mathbb{C}$ and $\mathbb{D}$ Categories with Families such that $\square_F$ and $\square_G$ exist. If (A, in) is initial in $\mathsf{Dialg}(F, G)$ then the elimination principle holds for (A, in).

Proof. Let $P \in \mathrm{Ty}(X)$ and $g \in (F(X)\cdot\square_F(P) \vdash \square_G(P)[\mathsf{in}\circ\mathbf{p}])$ be given. Then $h := \varphi_G^{-1} \circ \langle \mathsf{in}\circ\mathbf{p}, g\rangle \circ \varphi_F : F(X\cdot P) \to G(X\cdot P)$, so by initiality, we have a morphism $\mathsf{fold}(h) : X \to X\cdot P$ such that $h \circ F(\mathsf{fold}(h)) = G(\mathsf{fold}(h)) \circ \mathsf{in}$. Hence the following diagram commutes:

$$\begin{tikzcd}
F(X) \arrow[rrr, "\mathsf{in}"] \arrow[d, "F(\mathsf{fold}(h))"'] & & & G(X) \arrow[d, "G(\mathsf{fold}(h))"] \\
F(X\cdot P) \arrow[dr, "\varphi_F"'] \arrow[dd, "F(\mathbf{p})"'] & & & G(X\cdot P) \arrow[dl, shift right, "\varphi_G"'] \arrow[dd, "G(\mathbf{p})"] \\
& F(X)\cdot\square_F(P) \arrow[r, "\langle\mathsf{in}\circ\mathbf{p},g\rangle"] \arrow[dl, "\mathbf{p}"] & G(X)\cdot\square_G(P) \arrow[ur, shift right, "\varphi_G^{-1}"'] \arrow[dr, "\mathbf{p}"] & \\
F(X) \arrow[rrr, "\mathsf{in}"'] & & & G(X)
\end{tikzcd}$$

This means that $\mathbf{p}\circ\mathsf{fold}(h) : X \to X$ is a morphism in $\mathsf{Dialg}(F, G)$, so by initiality, we must have $\mathbf{p}\circ\mathsf{fold}(h) = \mathsf{id}$. We now define $\mathsf{elim}(P, g) := \mathbf{q}[\mathsf{fold}(h)]$. We then have

$$\mathsf{elim}(P, g) \in (X \vdash P[\mathbf{p}\circ\mathsf{fold}(h)]) = (X \vdash P[\mathsf{id}]) = (X \vdash P)$$

as required.

We must check that the computation rule $\overline{G}(\mathsf{elim}(P, g))[\mathsf{in}] = g[\overline{\overline{F}(\mathsf{elim}(P, g))}]$ holds. Note first that since $\mathbf{p}\circ\mathsf{fold}(h) = \mathsf{id}$, we have

$$\mathsf{fold}(h) = \langle \mathbf{p}\circ\mathsf{fold}(h), \mathbf{q}[\mathsf{fold}(h)]\rangle = \langle \mathsf{id}, \mathbf{q}[\mathsf{fold}(h)]\rangle = \overline{\mathbf{q}[\mathsf{fold}(h)]} = \overline{\mathsf{elim}(P, g)}$$

Using this, we have

$$\begin{aligned}
\overline{G}(\mathsf{elim}(P, g))[\mathsf{in}] &= \mathbf{q}[\varphi_G \circ G(\mathsf{fold}(h)) \circ \mathsf{in}] \\
&= \mathbf{q}[\varphi_G \circ \varphi_G^{-1} \circ \langle \mathsf{in}\circ\mathbf{p}, g\rangle \circ \varphi_F \circ F(\mathsf{fold}(h))] \\
&= g[\varphi_F \circ F(\mathsf{fold}(h))] \\
&= g[\varphi_F \circ F(\overline{\mathsf{elim}(P, g)})] \\
&= g[\overline{\overline{F}(\mathsf{elim}(P, g))}]
\end{aligned}$$

where we have used Lemma 4.32 in the last line. □

4.3.2 The elimination rules imply initiality

In Type Theory, one can show that inductive types are weakly initial by using the elimination rules with a non-dependent, constant motive. This proof also carries over to the generic setting, as long as the Categories with Families involved have constant families that interact well with $\square_G$.

Theorem 4.42 Let $F, G : \mathbb{C} \to \mathbb{D}$ with $\mathbb{C}$ and $\mathbb{D}$ CwFs with constant families, such that $\widetilde{G(\Delta)} = \Box_G(\check{\Delta})$ and $\overline{G}(M) = G(M^{\uparrow})^{\downarrow}$ (the second equation type checks because of the first). Let (X, in) be an (F, G)-dialgebra. If the elimination principle holds for (X, in), then (X, in) is weakly initial in $\mathsf{Dialg}(F, G)$.

Proof. Let (B, f) be an (F, G)-dialgebra. We have to construct $\mathsf{fold}(f) : X \to B$ such that $G(\mathsf{fold}(f)) \circ \mathsf{in} = f \circ F(\mathsf{fold}(f))$.

Notice that $\mathbf{q}_{\check{B}} \in (X \cdot \check{B}_X \vdash \check{B}_X[\mathbf{p}]) = (X \cdot \check{B}_X \vdash \check{B}_{X \cdot \check{B}})$ so that $\mathbf{q}^{\uparrow} : X \cdot \check{B} \to B$. Hence we have $\psi := f \circ F(\mathbf{q}^{\uparrow}) \circ \varphi_F^{-1} : F(X) \cdot \Box_F(\check{B}) \to G(B)$. Since $\widetilde{G(B)}_{F(X) \cdot \Box_F(\check{B})} = \widetilde{G(B)}_{G(X)}[\mathsf{in} \circ \mathbf{p}] = \Box_G(\check{B})[\mathsf{in} \circ \mathbf{p}]$, we then have $\psi^{\downarrow} \in (F(X) \cdot \Box_F(\check{B}) \vdash \Box_G(\check{B})[\mathsf{in} \circ \mathbf{p}])$ so that $\mathsf{elim}(\check{B}, \psi^{\downarrow}) \in (X \vdash \check{B})$. Hence we define $\mathsf{fold}(f) := \mathsf{elim}(\check{B}, \psi^{\downarrow})^{\uparrow} : X \to B$. We now check that the diagram commutes:

$$\begin{aligned}
G(\mathsf{fold}(f)) \circ \mathsf{in} &= G(\mathsf{elim}(\check{B}, \psi^{\downarrow})^{\uparrow}) \circ \mathsf{in} \\
&= \overline{G}(\mathsf{elim}(\check{B}, \psi^{\downarrow}))[\mathsf{in}]^{\uparrow} \\
&= \psi^{\downarrow}[\overline{\overline{F}(\mathsf{elim}(\check{B}, \psi^{\downarrow}))}]^{\uparrow} \\
&= \psi \circ \overline{\overline{F}(\mathsf{elim}(\check{B}, \psi^{\downarrow}))} \\
&= f \circ F(\mathbf{q}^{\uparrow}) \circ \varphi_F^{-1} \circ \varphi_F \circ F(\overline{\mathsf{elim}(\check{B}, \psi^{\downarrow})}) \\
&= f \circ F(\mathbf{q}^{\uparrow} \circ \overline{\mathsf{elim}(\check{B}, \psi^{\downarrow})}) \\
&= f \circ F(\mathbf{q}[\overline{\mathsf{elim}(\check{B}, \psi^{\downarrow})}]^{\uparrow}) \\
&= f \circ F(\mathsf{elim}(\check{B}, \psi^{\downarrow})^{\uparrow}) = f \circ F(\mathsf{fold}(f))
\end{aligned}$$

□

For $G = \mathsf{Id} : \mathbb{C} \to \mathbb{C}$, Lemma 4.21 ($\Box_{\mathsf{Id}} = \mathsf{Id}$) and Lemma 4.32 ($\overline{\mathsf{Id}} = \mathsf{Id}$) ensures that the conditions on the constant families are satisfied, since $\widetilde{\mathsf{Id}(\Delta)} = \check{\Delta} = \Box_{\mathsf{Id}}(\check{\Delta})$ and $\overline{\mathsf{Id}}(M) = M = (M^{\uparrow})^{\downarrow} = \mathsf{Id}(M^{\uparrow})^{\downarrow}$.

For the functors arising from the axiomatisation in Chapter 3, we can show that we in fact have strong initiality, since we can do induction over the codes. The proof relies on extensional equality, as we are working in extensional Type Theory in this chapter.

Theorem 4.43 Let $\mathsf{Arg}_{\gamma} : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$ be a functor representing an inductive-inductive definition. Let $(A, B, \mathsf{in}_A, \mathsf{in}_B) \in \mathbb{E}_{\mathsf{Arg}_{\gamma}}$. The general elimination principle holds for $(A, B, \mathsf{in}_A, \mathsf{in}_B)$ if and only if $(A, B, \mathsf{in}_A, \mathsf{in}_B)$ is initial in $\mathbb{E}_{\mathsf{Arg}_{\gamma}}$.

Proof. After applying Theorem 4.41 and Theorem 4.42, and discharging the extra assumptions on $V : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$, all that is left to be proven is that if the elimination principle holds, not only is $(A, B, \mathsf{in}_A, \mathsf{in}_B)$ weakly initial, but $\mathsf{fold}(f, f') := (\mathsf{fold}(f), \mathsf{fold}(f'))$ is in fact unique, where $(f, f') : \mathsf{Arg}_{\gamma}(A', B') \to (A', B')$ is the morphism of another dialgebra. Let $(h, g) : (A, B, \mathsf{in}_A, \mathsf{in}_B) \to (A', B', f, f')$ be another morphism that makes the diagram commute. We prove $h(x) = \mathsf{fold}(f)_0(x)$ and $g(x, y) = \mathsf{fold}(f)_1(x, y)$ for all $x : A$ and $y : B(x)$ by applying the elimination principle with $P(x) = h(x) \equiv_{A'} \mathsf{fold}(f)_0(x)$ and $Q(x, y, \widetilde{x}) = g(x, y) \equiv_{B'(h(x))} \mathsf{fold}(f)_1(x, y)$. Notice that this is type correct because of the argument $\widetilde{x} : h(x) \equiv_{A'} \mathsf{fold}(f)_0(x)$ and extensional equality: $\mathsf{fold}(f)_1(x, y)$ is of type $B'(\mathsf{fold}(f)_0(x))$.

Thus, it is enough to prove P and Q for canonical inhabitants, given that the equations hold for subterms. We need to find $\mathsf{step}_{\mathsf{in}_A}(x, \widetilde{x}) : P(\mathsf{in}_A(x))$, i.e. prove $h(\mathsf{in}_A(x)) \equiv_{A'} \mathsf{fold}(f)_0(\mathsf{in}_A(x))$ given $\widetilde{x} : \Box_{\mathsf{Arg_A}}(P, x)$, and similarly find $\mathsf{step}_{\mathsf{in}_B}(\ldots) : Q(\mathsf{in}_A(x), \mathsf{in}_B(x, y), \mathsf{step}_{\mathsf{in}_A}(x, \widetilde{x}))$. But this is now straightforward by induction over the codes: in the base case nil, the result follows from the fact that (h, g) makes the diagram commute, and in the step cases, the result follows immediately by the induction hypothesis. □

4.4 Summary and discussion

In this chapter, we have introduced a general categorical framework for describing elimination rules, and then instantiated it for inductive-inductive definitions by modelling them as certain dialgebras. This gives a less syntactic view of inductive-inductive definitions. Next, we have proven that the elimination rules in this general setting, of which the elimination rules for inductive-inductive definitions are an instance, are equivalent to a more categorical notion: that a certain object is initial in a category of dialgebras.

Why so general? There are two reasons for adopting the abstract framework as we have done in this chapter. First and foremost, it is for our own sanity; by keeping things abstract, we can work with shorter equations and get away with keeping track of less details. However, there is also a technical reason for proving say Theorem 4.37 at the level of generality that we did. In Chapter 6, we will extend the current theory in order to handle more complex situations such as e.g. adding a third simultaneously defined data type $C : (a : A) \to B(a) \to \mathsf{Set}$. The theorems of this chapter will immediately scale and apply also in this setting.

Initial algebras for degenerate inductive-inductive definitions The need to introduce the more complicated machinery of dialgebras comes from the fact that type of the constructor $\mathsf{intro_B}$ for $B : A \to \mathsf{Set}$ can contain the constructor $\mathsf{intro_A}$ for the index set A. If this does not occur, we can get away with using ordinary algebras for endofunctors on $\mathsf{Fam}(\mathsf{Set})$. More or less by definition, if the constructor $\mathsf{intro_A}$ is never used by a functor $\mathsf{Arg} : \mathsf{Dialg}(\mathsf{Arg_A}, U) \to \mathsf{Fam}(\mathsf{Set})$, then this functor is really an endofunctor $\mathsf{Arg} : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Fam}(\mathsf{Set})$ and the usual, well-known theory of initial algebras apply. Moreover, one can check that the elimination principle one gets coincides with the expected one.

Instantiating the framework The dialgebraic framework presented in this chapter can be instantiated to many known classes of data types, some of which are collected in Table 4.1. The functor $F : \mathbb{C} \to \mathbb{D}$ is used to describe the concrete data type, while the parameters $\mathbb{C}$, $\mathbb{D}$ and $G : \mathbb{C} \to \mathbb{D}$ are fixed. We have already hinted at that Set and

Table 4.1: Instances of the dialgebraic framework.

Data types	$\mathbb{C}$	$\mathbb{D}$	G
Inductive definitions	Set	Set	id
Indexed inductive definitions	Set^I	Set^I	id
Inductive-recursive definitions[a]	type/D	type/D	id
inductive-inductive definitions[b]	$\mathsf{Dialg}(\mathsf{Arg}_\mathsf{A}, U)$	Fam(Set)	V

[a] Modulo size issues. [b] In a subcategory.

Set^I can be used for inductive and indexed inductive definitions respectively. Dybjer and Setzer [2003] give an initial algebra semantics for inductive-recursive definitions, using slice categories, which coincides with our dialgebraic semantics presented here when instantiated to the same slice category. Finally, we see that inductive-inductive definitions are the only example to date where dialgebras are used instead of ordinary F-algebras.

Related work Another closely related framework for generic induction/elimination rules is the framework described by Hermida and Jacobs [1998], later extended by Ghani et al. [2010, 2011] (see also Fumex [2012]). Their idea is to model types by objects in a category $\mathbb{B}$, and properties by the total category of a fibration $p : \mathbb{E} \to \mathbb{B}$; the functor p maps each property to the type it is a property of. By asking for some extra structure, namely a comprehension category with unit, induction principles can be given an elegant characterisation in terms of liftings of functors and adjoint equivalences.

Given a Category with Families with constant families, one can construct a split full comprehension category with unit (see Jacobs [1999, Exercise 10.4.6]), and conversely, given a split comprehension category with units, one can construct a Category with Families. We see that there is a slight gap: constant families are needed to get a unit, but a unit does not necessarily give constant families. Furthermore, starting from a comprehension category, one might not get a $\Box_F$ type in general.

The fibrational approach is hence in one sense more general. On the other hand, it also tries to do less. First of all, only endofunctors and ordinary algebras are considered, not arbitrary functors and dialgebras, so the framework is not suitable for inductive-inductive definitions. Furthermore, no equivalence between induction principles and initiality is shown, only that if an initial algebra exists in a certain category, then the induction principle is valid. Another advantage of our Categories with Families framework is that we can also talk about the computation rules associated with the eliminators, something that is missing in the fibrational setting.

Chapter 5

Semantics

Contents

So far, we have given an axiomatisation of a type theory with inductive-inductive definitions, together with a more streamlined categorical characterisation using extensional Type Theory. In this chapter, we first prove that our theory is consistent by constructing a "standard" set-theoretic model in Section 5.1. We then give a second consistency proof by interpreting the theory in the (extensional) theory of indexed inductive definitions in Section 5.3, making use of a kind of "container normal form" for inductive-inductive definitions, which is developed in Section 5.2. This also provides a tighter bound for the proof-theoretic strength of the theory.

A shorter description of the model described in Section 5.1 has previously appeared in the proceedings of CSL 2010 [Nordvall Forsberg and Setzer, 2010] and the Schwichtenberg Festschrift [Nordvall Forsberg and Setzer, 2012].

5.1 A set-theoretic model

We will develop a model in ZFC set theory, extended by two inaccessible cardinals in order to interpret Set and large types. The main feature of the model is that it is natural and straightforward: types are interpreted as sets, terms as elements, the typing relation $x : A$ as the membership relation $x \in A$ etc. Inductive-inductive definitions are interpreted by iterating a monotone operator until a fixed point is reached. Our model is a simpler version of those developed by Dybjer and Setzer [1999, 2006] for induction-recursion. See Aczel [1999] for a more detailed treatment of interpreting Type Theory in set theory.

5.1.1 Dialgebras versus F-algebras

A priori, we need to construct a model which validates the dependent (general) elimination rules from Section 3.2.5. However, by Theorem 4.43, it is enough to construct an initial dialgebra, i.e. validate non-dependent elimination. We now make a simple observation that will make life even easier for us: if we happen to be so lucky that the functor G has a left adjoint $L \dashv G$, then we can equivalently consider $(L \circ F)$-algebras instead of (F, G)-dialgebras.

Lemma 5.1 Let $F, G : \mathbb{C} \to \mathbb{D}$ be functors, such that G has a left adjoint $L : \mathbb{D} \to \mathbb{C}$. The categories $\mathsf{Alg}_{L \circ F}$ and $\mathsf{Dialg}(F, G)$ are isomorphic.

Proof. The natural isomorphisms $\phi_{X,A} : \mathsf{Hom}(L(X), A) \to \mathsf{Hom}(X, G(A))$ between hom-sets induce an isomorphism between $\mathsf{Alg}_{L \circ F}$ and $\mathsf{Dialg}(F, G)$, which sends $(X, h : L(F(X)) \to X)$ in $\mathsf{Alg}_{L \circ F}$ to $(X, \phi_{F(X),X}(h))$ and $(X, k : F(X) \to G(X))$ in $\mathsf{Dialg}(F, G)$ to $(X, \phi^{-1}_{F(X),X}(k))$. The functors are identities on morphisms. The required squares commute because of the naturality of $\phi_{X,A}$: For any $f : X' \to X$ and $g : A \to A'$, we have

$$\phi_{X',A'}(g \circ h \circ L(f)) = G(g) \circ \phi_{X,A}(h) \circ f$$

In detail, if $g : (X, h) \to (Y, h')$ in $\mathsf{Alg}_{L \circ F}$, we need to check that also $g : (X, \phi(h)) \to (Y, \phi(h'))$ in $\mathsf{Dialg}(F, G)$. In other words, we need to check that the right hand side diagram commutes given that the left hand side does:

$$\begin{tikzcd}
L(F(X)) \arrow[r, "h"] \arrow[d, "L(F(g))"'] & X \arrow[d, "g"] \\
L(F(Y)) \arrow[r, "h'"'] & Y
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
F(X) \arrow[r, "\phi(h)"] \arrow[d, "F(g)"'] & G(X) \arrow[d, "G(g)"] \\
F(Y) \arrow[r, "\phi(h')"'] & G(Y)
\end{tikzcd}$$

But this is straightforward:

$$G(g) \circ \phi(h) = \phi(g \circ h) = \phi(h' \circ L(F(g))) = \phi(h') \circ F(g)$$

The other direction works in exactly the same way, using the naturality of $\phi^{-1}_{X,A}$:

$$g \circ \phi^{-1}_{X,A}(k) \circ L(f) = \phi^{-1}_{X',A'}(G(g) \circ k \circ f) \enspace .$$

Finally, since $\phi_{F(X),X}$ is an isomorphism, the induced functor obviously is too. □

In particular, in the situation of the proposition, $\mathsf{Alg}_{L \circ F}$ has an initial object if and only if $\mathsf{Dialg}(F, G)$ does. Inspecting the proof, we see that the carriers of the algebras are preserved by the isomorphism, so that the carrier of the initial algebra in $\mathsf{Alg}_{L \circ F}$ is the carrier of the initial dialgebra in $\mathsf{Dialg}(F, G)$.

When describing inductive-inductive definitions, dialgebras crop up in two places: first when describing the domain of the functor $\mathsf{Arg} : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$, and secondly in the supercategory $\mathsf{Dialg}(\mathsf{Arg}, V)$ of the category we are interested in. We will now see that both $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ and $V : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$ in fact have left adjoints.

Proposition 5.2 The index set functor $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ has a left adjoint $L : \mathsf{Set} \to \mathsf{Fam}(\mathsf{Set})$ given by $L(X) = (X, \lambda x.\, 0)$ and $L(f) = (f, \lambda x.\, \mathsf{id})$.

Proof. It is easy to see that we have a bijective correspondence between the hom-sets $\mathsf{Hom}(L(X), (A,B))$ and $\mathsf{Hom}(X, U(A,B)))$:

$$\begin{array}{c} (f,g) : L(X) \to (A,B) \\ \hline\hline f : X \to A \quad g_x : 0 \to B(x) \\ \hline\hline f : X \to A \\ \hline\hline f : X \to U(A,B) \end{array}$$

Naturality is also easily dealt with. □

We now show that also $V : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U) \to \mathsf{Fam}(\mathsf{Set})$ (or, equivalently, $V : \mathsf{Alg}_{L \circ \mathsf{Arg}_{\mathsf{A}}} \to \mathsf{Fam}(\mathsf{Set})$) has a left adjoint, which will be a free algebra in the following sense:

Definition 5.3 Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor and Y an object in $\mathbb{C}$. A free F-algebra on Y is an F-algebra (X, h) together with a morphism $\eta_Y : Y \to X$ in $\mathbb{C}$ such that for any F-algebra (A, a) and morphism $f : Y \to A$, there exists a unique F-algebra morphism $f^{\dagger} : (X, h) \to (A, a)$ such that the following diagram commutes:

$$\begin{tikzcd} F(X) \arrow[r, "h"] \arrow[d, "F(f^{\dagger})"'] & X \arrow[d, "f^{\dagger}"'] & Y \arrow[l, "\eta_Y"'] \arrow[dl, "f"] \\ F(A) \arrow[r, "a"] & A & \end{tikzcd}$$

Lemma 5.4 (Adámek [1974]) The carrier of the initial algebra for the functor $F(-) + Y$ is the carrier of a free F-algebra on Y and vice versa. In particular, the free F-algebra on Y is unique up to isomorphism.

Proof. By the universal property of coproducts, a morphism $F(X) + Y \to X$ corresponds exactly to a morphism $F(X) \to X$ and a morphism $Y \to X$. An easy diagram chase then confirms that all the necessary diagrams commute.

In detail, let (X, in) be the initial $F(-) + Y$-algebra. Then $(X, \mathsf{in} \circ \mathsf{inl})$ is an F-algebra, and we can define $\eta_Y := \mathsf{in} \circ \mathsf{inr}$. Given any F-algebra (A, a) and morphism $f : Y \to A$, we can construct an algebra $(A, [a, f])$ for the functor $F(-) + Y$, and we can define $f^{\dagger}$ as $f^{\dagger} := \mathsf{fold}([a, f])$. The necessary diagram commutes since the corresponding diagram for $\mathsf{fold}([a, f])$ does, and also uniqueness follows from uniqueness of $\mathsf{fold}([a, f])$ and the universal property of coproducts. The other direction follows in the same way. □

This gives us a way to compute the free F-algebra on Y; we consider the initial sequence

$$0 \to F(0) + Y \to F(F(0) + Y) + Y \to \ldots$$

of $F(-) + Y$ and show that it converges at some stage α.

Definition 5.5 (Free H-algebra functor) Let $H : \mathbb{C} \to \mathbb{C}$ be a functor such that free H-algebras exist on every object. The *free H-algebra functor* $F : \mathbb{C} \to \mathsf{Alg}_H$ sends Y in $\mathbb{C}$ to the free H-algebra on Y. The action on morphisms is induced by the freeness of the algebra: If $f : Y \to Y'$ is a morphism in $\mathbb{C}$, $F(f) = (\eta_{Y'} \circ f)^\dagger$. ◼

Proposition 5.6 The free H-algebra functor $F : \mathbb{C} \to \mathsf{Alg}_H$ is left adjoint to the forgetful functor $U : \mathsf{Alg}_H \to \mathbb{C}$.

Proof. We have to check that morphisms $f : F(Y) \to (X, h)$ in Alg_H are in bijective correspondence with morphisms $g : Y \to X$ in $\mathbb{C}$. Let us write $F(Y) = ((\overline{Y}, k), \eta_Y)$. Given $f : F(Y) \to (X, h)$, we have $f \circ \eta_Y : Y \to X$, and given $g : Y \to X$, by definition $g^\dagger : \overline{Y} \to X$. We check that these constructions are inverse to each other. That $g^\dagger \circ \eta_Y = g$ is exactly the fact that the right hand triangle in Definition 5.3 commutes. The fact that $(f \circ \eta_Y)^\dagger = f$ is less immediate, but follows from the uniqueness of $(f \circ \eta_Y)^\dagger$ and the fact that $f \circ k = h \circ H(f)$ as a morphism in Alg_F. A routine verification shows naturality in Y and (X, h). □

Thus, in particular the forgetful functor $V : \mathsf{Alg}_{L \circ \mathsf{Arg}_\mathrm{A}} \to \mathsf{Fam}(\mathsf{Set})$ has a left adjoint $M : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Alg}_{L \circ \mathsf{Arg}_\mathrm{A}}$ and, by Lemma 5.1, the category $\mathsf{Dialg}(\mathsf{Arg}, V)$ is isomorphic to $\mathsf{Alg}_{M \circ \mathsf{Arg}}$. Hence the subcategory $\mathbb{E}_{\mathsf{Arg}}$ is isomorphic to a subcategory of $\mathsf{Alg}_{M \circ \mathsf{Arg}}$.

Theorem 5.7 For each functor $\mathsf{Arg} = (\mathsf{Arg}_\mathrm{A}, \mathsf{Arg}_\mathrm{B})$ representing an inductive-inductive definition, $\mathbb{E}_{\mathsf{Arg}}$ has an initial object.

Proof. By the results from this section, we need to find an initial $(M \circ \mathsf{Arg})$-algebra, where $M : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Alg}_{L \circ \mathsf{Arg}_\mathrm{A}}$ is the free $L \circ \mathsf{Arg}_\mathrm{A}$-algebra functor. The functor M is well-defined since Arg_A is "strictly positive" by construction: arguments X never occur to the left of a function arrow in $\mathsf{Arg}_\mathrm{A}(X)$. Hence the size κ of all premises of inductive arguments is independent of X, and the initial sequence for the functor $L(\mathsf{Arg}_\mathrm{A}(-) + X$ (hence for $M(X)$ by Proposition 5.6) will converge after κ^+ iterations by a generalisation of Proposition 1.6, where κ^+ is the least regular cardinal above κ.

In the same way, also Arg_B is also strictly positive, and hence the initial sequence

$$\mathbf{0} \to M(\mathsf{Arg}(\mathbf{0})) \to M(\mathsf{Arg}(M(\mathsf{Arg}(\mathbf{0})))) \to \ldots$$

will converge, again by an argument similar to Proposition 1.6. Hence an initial $(M \circ \mathsf{Arg})$-algebra exists [Adámek et al., 2010, Thm 3.1.4]. □

5.1.2 A concrete model

For completeness, we now use Theorem 5.7 to give a concrete model in ZFC set theory, extended with two inaccessible cardinals to interpret Set and large types.

5.1.2.1 Preliminaries set theory

We recall some standard definitions and properties that we will use in the model construction. We employ classical logic in this section.

Definition 5.8 (Regular and inaccessible) Let κ be a cardinal.

(i) κ is *regular* if the cofinality of κ is κ, i.e. if $\sup f < \kappa$ for all strictly increasing functions $f : \alpha \to \kappa$ with $\alpha < \kappa$.

(ii) κ is *inaccessible* if it is regular and a strong limit cardinal (i.e. if $\beta < \kappa$ then $2^\beta < \kappa$). It is *weakly inaccessible* if it is regular and a limit cardinal (i.e. if $\beta < \kappa$ then $\beta^+ < \kappa$, where β^+ is the least cardinal larger than β). ■

Note that all regular cardinals are limit *ordinals*, but not necessarily limit *cardinals*. For simplicity, we will assume the generalised continuum hypothesis, which states that $\beta^+ = 2^\beta$. Thus, under this hypothesis, weak and strong inaccessibility coincides.

Definition 5.9 (The cumulative hierarchy) The cumulative hierarchy V_α is a collection of sets indexed by ordinals α, defined by transfinite recursion as follows:

$$\begin{aligned} V_0 &= \varnothing \\ V_{\beta+1} &= \mathcal{P}(V_\beta) \\ V_\lambda &= \bigcup_{\beta<\lambda} V_\beta \quad \text{for } \lambda \text{ limit} \end{aligned}$$ ■

One can check that $V_\alpha = \bigcup_{\beta<\alpha} \mathcal{P}(V_\beta)$ for every ordinal α. We will often make use of the following:

Proposition 5.10

(i) V_α is transitive: if $x \in V_\alpha$ then $x \subseteq V_\alpha$.

(ii) V_α is closed under subsets: if $x \in V_\alpha$ and $y \subseteq x$ then $y \in V_\alpha$.

(iii) V_α is monotone: if $\alpha < \beta$, then $V_\alpha \subseteq V_\beta$.

(iv) If $x \in V_\kappa$ with κ inaccessible then $|x| < \kappa$. □

5.1.2.2 Interpretation of expressions

We will be working informally in ZFC extended with the existence of two strongly inaccessible cardinals $\mathfrak{i}_0 < \mathfrak{i}_1$. We use standard set theoretic constructions, e.g.

$$\begin{aligned} \langle a,b\rangle &:= \{\{a\},\{a,b\}\} \ , \\ \lambda x \in a \,.\, b(x) &:= \{\langle x, b(x)\rangle \mid x \in a\} \ , \\ \Pi_{x\in a} b(x) &:= \{f : a \to \bigcup_{x\in a} b(x) \mid \forall x \in a. f(x) \in b(x)\} \ , \\ \Sigma_{x\in a} b(x) &:= \{\langle c,d\rangle \mid c \in a \wedge d \in b(c)\} \ , \\ 0 &:= \varnothing, 1 := \{0\}, 2 := \{0,1\} \ , \\ a_0 + \ldots + a_n &:= \Sigma_{i\in\{0,\ldots,n\}} a_i \end{aligned}$$

Whenever we introduce sets A^α indexed by ordinals α, let

$$A^{<\alpha} := \bigcup_{\beta<\alpha} A^\beta \ .$$

For every expression A of our type theory, we will give an interpretation $[\![A]\!]_\rho$, regardless of whether A type or $A : B$ or not. Interpretations might however be undefined, written $[\![A]\!]_\rho \uparrow$. If $[\![A]\!]_\rho$ is defined, we write $[\![A]\!]_\rho \downarrow$. We write $A \simeq B$ for partial equality, i.e. $A \simeq B$ if and only if $A \downarrow \Leftrightarrow B \downarrow$ and if $A \downarrow$, then $A = B$. We write $A :\simeq B$ if we define A such that $A \simeq B$.

Open terms will be interpreted relative to an environment ρ, i.e. a function mapping variables to terms. Write $\rho_{[x\mapsto a]}$ for the environment ρ extended with $x \mapsto a$, i.e. $\rho_{[x\mapsto a]}(y) = a$ if $y = x$ and $\rho(y)$ otherwise. The interpretation $[\![t]\!]_\rho$ of closed terms t will not depend on the environment, and we omit the subscript ρ.

The interpretation of the logical framework is as in Dybjer and Setzer [1999]:

$$[\![\mathsf{Set}]\!] := V_{\mathrm{i}_0} \qquad [\![\mathsf{type}]\!] := V_{\mathrm{i}_1}$$

$$[\![(x:A)\to B]\!]_\rho :\simeq \Pi_{y\in[\![A]\!]_\rho}[\![B]\!]_{\rho_{[y\mapsto x]}} \qquad [\![\lambda(x:A).\,e]\!]_\rho :\simeq \lambda y\in[\![A]\!]_\rho\,.\,[\![e]\!]_{\rho_{[y\mapsto x]}}$$

$$[\![(\Sigma x:A)B]\!]_\rho :\simeq \Sigma_{y\in[\![A]\!]_\rho}[\![B]\!]_{\rho_{[y\mapsto x]}} \qquad [\![\langle a,b\rangle]\!]_\rho :\simeq \langle [\![a]\!]_\rho, [\![b]\!]_\rho\rangle$$

$$[\![\mathbf{0}]\!] := 0 \qquad [\![\mathbf{1}]\!] := 1 \qquad [\![\mathbf{2}]\!] := 2 \qquad [\![\star]\!] := 0 \qquad [\![\mathsf{tt}]\!] := 0 \qquad [\![\mathsf{ff}]\!] := 1$$

$$[\![\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b]\!]_\rho :\simeq \begin{cases} [\![a]\!]_\rho & \text{if } [\![x]\!]_\rho = 0 \\ [\![b]\!]_\rho & \text{if } [\![x]\!]_\rho = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![!_A]\!]_\rho :\simeq \varnothing \text{ (the unique inclusion } \varnothing \to [\![A]\!]_\rho\text{)}$$

Note that we interpret large elimination for **2** (at no extra cost). The η rules are easily proved.

To interpret terms containing SP_A, SP_B, Arg_A, Arg_B, $\mathsf{Index}_\mathrm{B}$, and the codes nil, non-ind, A-ind and B-ind, we first define $[\![\mathsf{SP}_\mathrm{A}]\!]$, $[\![\mathsf{SP}_\mathrm{B}]\!]$, $[\![\mathsf{Arg}_\mathrm{A}]\!]$, $[\![\mathsf{nil}]\!]$, $[\![\mathsf{non\text{-}ind}]\!]$, ...and interpret

$$[\![\mathsf{SP}_\mathrm{A}(X_{\mathrm{ref}})]\!]_\rho := [\![\mathsf{SP}_\mathrm{A}]\!]([\![X_{\mathrm{ref}}]\!]_\rho)$$

$$\vdots$$

$$[\![\mathsf{Arg}_\mathrm{A}(X_{\mathrm{ref}},\gamma,X,Y,\mathsf{rep}_\mathrm{X})]\!]_\rho := [\![\mathsf{Arg}_\mathrm{A}]\!]([\![X_{\mathrm{ref}}]\!]_\rho, [\![\gamma]\!]_\rho, [\![X]\!]_\rho, [\![Y]\!]_\rho, [\![\mathsf{rep}_\mathrm{X}]\!]_\rho)$$

$$\vdots$$

$$[\![\mathsf{non\text{-}ind}(K,\gamma)]\!]_\rho := [\![\mathsf{non\text{-}ind}]\!]([\![K]\!]_\rho, [\![\gamma]\!]_\rho)$$

$$\vdots \ \text{ etc.}$$

In all future definitions, if we are currently defining $[\![F]\!]_\rho$ where $F : D \to E$, say, let $[\![F]\!]_\rho(d)\uparrow$ if $d \notin [\![D]\!]_\rho$.

$\llbracket \mathsf{SP}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}})$ is defined as the least set such that

$$\begin{aligned}\llbracket \mathsf{SP}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}) = 1 + \sum_{K \in \llbracket \mathsf{Set} \rrbracket} (K \to \llbracket \mathsf{SP}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}})) + \sum_{K \in \llbracket \mathsf{Set} \rrbracket} \llbracket \mathsf{SP}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}} + K) \\ + \sum_{K \in \llbracket \mathsf{Set} \rrbracket} \sum_{h : K \to X_{\mathrm{ref}}} \llbracket \mathsf{SP}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}) \ .\end{aligned}$$

Such a set exists by the inaccessibility of i_0. The constructors are then interpreted as

$$\begin{aligned}\llbracket \mathsf{nil} \rrbracket :\simeq \langle 0, 0 \rangle \quad \llbracket \mathsf{B\text{-}ind} \rrbracket(K, h, \gamma) :\simeq \langle 3, \langle K, \langle h, \gamma \rangle \rangle \rangle \\ \llbracket \mathsf{non\text{-}ind} \rrbracket(K, \gamma) := \langle 1, \langle K, \gamma \rangle \rangle \quad \llbracket \mathsf{A\text{-}ind} \rrbracket(K, \gamma) :\simeq \langle 2, \langle K, \gamma \rangle \rangle\end{aligned}$$

$\llbracket \mathsf{SP}_{\mathrm{B}} \rrbracket$ and its constructors are defined analogously. The functions $\llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket$, $\llbracket \mathsf{Arg}_{\mathrm{B}} \rrbracket$ and $\llbracket \mathsf{Index}_{\mathrm{B}} \rrbracket$ are defined according to their equations, e.g.

$$\begin{aligned}\llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}, \llbracket \mathsf{nil} \rrbracket, X, Y, \mathsf{rep}_{\mathrm{X}}) &:\simeq 1 \\ \llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}, \llbracket \mathsf{non\text{-}ind} \rrbracket(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &:\simeq \sum_{k \in K} \llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}, \gamma(k), X, Y, \mathsf{rep}_{\mathrm{X}}) \\ \llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}, \llbracket \mathsf{A\text{-}ind} \rrbracket(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &:\simeq \sum_{j : K \to A} \llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}} + K, \gamma, X, Y, [\mathsf{rep}_{\mathrm{X}}, j]) \\ \llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}, \llbracket \mathsf{B\text{-}ind} \rrbracket(K, h, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &:\simeq \sum_{j \in \Pi_{k \in K} B(\mathsf{rep}_{\mathrm{X}}(h(k)))} \llbracket \mathsf{Arg}_{\mathrm{A}} \rrbracket(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}}) .\end{aligned}$$

Finally, we have to interpret $A_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}$, $B_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}$, $\mathsf{intro}_{A_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}}$ and $\mathsf{intro}_{B_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}}$, for which we use Theorem 5.7. Concretely, this means that we iterate $\mathsf{Arg}^0_{\mathrm{A}}$ until a fixed point is reached, then apply $\mathsf{Arg}^0_{\mathrm{B}}$ once, and repeat. This is intuitively necessary since $\mathsf{Arg}^0_{\mathrm{B}}$ expects an argument $\mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, A, B) \to A$, which can be chosen to be the identity if A is a fixed point of $\mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, A, B)$ (with B fixed). In more detail, let

$$\begin{aligned}\llbracket A_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}} \rrbracket :\simeq A^{\mathrm{i}_0} \ , \quad \llbracket B_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}} \rrbracket(a) :\simeq B^{\mathrm{i}_0}(a) \ , \\ \llbracket \mathsf{intro}_{A_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}} \rrbracket(a) :\simeq a \ , \quad \llbracket \mathsf{intro}_{B_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}} \rrbracket(b) :\simeq b \ ,\end{aligned}$$

where A^α and B^α are simultaneously defined by recursion on α as

$$\begin{aligned}A^\alpha &:= \text{least fixed point containing } A^{<\alpha} \text{ of } \lambda X . \llbracket \mathsf{Arg}^0_{\mathrm{A}} \rrbracket(\gamma_A, X, B^{<\alpha}) \ , \\ B^\alpha(a) &:= \{ b \mid b \in \llbracket \mathsf{Arg}^0_{\mathrm{B}} \rrbracket(\gamma_A, A^\alpha, B^{<\alpha}, \mathsf{id}, \gamma_B) \\ &\qquad \wedge \llbracket \mathsf{Index}^0_{\mathrm{B}} \rrbracket(\gamma_A, A^\alpha, B^{<\alpha}, \mathsf{id}, \gamma_B, b) = a \} \ .\end{aligned}$$

The (graph of the) eliminators can then be built up in the same stages.

Having interpreted all terms, we finally interpret contexts as sets of environments:

$$\llbracket \varnothing \rrbracket :\simeq \varnothing \qquad \llbracket \Gamma, x : A \rrbracket :\simeq \{ \rho_{[x \mapsto a]} \mid \rho \in \llbracket \Gamma \rrbracket \wedge a \in \llbracket A \rrbracket_\rho \}.$$

5.1.2.3 Soundness of the rules

The verification of most of the rules is routine. The main difficulty lies in proving that $[\![\mathsf{SP}_\mathrm{A}]\!]$ and $[\![\mathsf{SP}_\mathrm{B}]\!]$ are well-defined, and that $[\![A_{\gamma_A,\gamma_B}]\!] \in [\![\mathsf{Set}]\!]$ and $[\![B_{\gamma_A,\gamma_B}]\!] : [\![A_{\gamma_A,\gamma_B}]\!] \to [\![\mathsf{Set}]\!]$.

$[\![\mathsf{SP}_\mathrm{A}]\!]$ is obtained by iterating the appropriate operator $\Gamma : ([\![\mathsf{Set}]\!] \to [\![\mathsf{Set}]\!]) \to ([\![\mathsf{Set}]\!] \to [\![\mathsf{Set}]\!])$ up to i_0 times. Since $X_\mathrm{ref} \in [\![\mathsf{Set}]\!]$, we have $(X_\mathrm{ref} + K)$, $(K \to X_\mathrm{ref}) \in [\![\mathsf{Set}]\!]$ for all $K \in [\![\mathsf{Set}]\!] = V_{\mathrm{i}_0}$ by the inaccessibility of i_0. Hence all "premises" have cardinality at most i_0, which is regular, so that the operator has a fixed point after i_0 iterations by Proposition 1.6. The fixed point must be an element of $[\![\mathsf{type}]\!] = V_{\mathrm{i}_1}$ by the inaccessibility of i_1.

To see that $[\![A_{\gamma_A,\gamma_B}]\!] \in [\![\mathsf{Set}]\!]$ and $[\![B_{\gamma_A,\gamma_B}]\!] : [\![A_{\gamma_A,\gamma_B}]\!] \to [\![\mathsf{Set}]\!]$, one first verifies that $[\![\mathsf{Arg}^0_\mathrm{A}]\!]$, $[\![\mathsf{Arg}^0_\mathrm{B}]\!]$, $[\![\mathsf{Index}^0_\mathrm{B}]\!]$ are monotone in the following sense:

Lemma 5.11 For all $\gamma_\mathrm{A} \in [\![\mathsf{SP}^0_\mathrm{A}]\!]$ and $\gamma_\mathrm{B} \in [\![\mathsf{SP}^0_\mathrm{B}]\!](\gamma_\mathrm{A})$:

(i) If $A \subseteq A'$ and $B(x) \subseteq B'(x)$ then $[\![\mathsf{Arg}^0_\mathrm{A}]\!](\gamma_\mathrm{A}, A, B) \subseteq [\![\mathsf{Arg}^0_\mathrm{A}]\!](\gamma_\mathrm{A}, A', B')$.

(ii) If in addition $\mathsf{intro}_\mathrm{A}(x) = \mathsf{intro}'_\mathrm{A}(x)$ for all $x \in \mathsf{Arg}^0_\mathrm{A}(\gamma_\mathrm{A}, A, B)$, then

$$[\![\mathsf{Arg}^0_\mathrm{B}]\!](\gamma_\mathrm{A}, A, B, \mathsf{intro}_\mathrm{A}, \gamma_\mathrm{B}) \subseteq [\![\mathsf{Arg}^0_\mathrm{B}]\!](\gamma_\mathrm{A}, A', B', \mathsf{intro}'_\mathrm{A}, \gamma_\mathrm{B})$$

and

$$[\![\mathsf{Index}^0_\mathrm{B}]\!](\gamma_\mathrm{A}, A, B, \mathsf{intro}_\mathrm{A}, \gamma_\mathrm{B}, x) = [\![\mathsf{Index}^0_\mathrm{B}]\!](\gamma_\mathrm{A}, A', B', \mathsf{intro}'_\mathrm{A}, \gamma_\mathrm{B}, x)$$

for all $x \in [\![\mathsf{Arg}^0_\mathrm{B}]\!](\gamma_\mathrm{A}, A, B, \mathsf{intro}_\mathrm{A}, \gamma_\mathrm{B})$. □

We can then adapt the standard results [Aczel, 1977] about monotone operators. First, we note that one application of $[\![\mathsf{Arg}^0_\mathrm{A}]\!]$ and $[\![\mathsf{Arg}^0_\mathrm{B}]\!]$ is not enough to take us outside of $[\![\mathsf{Set}]\!]$:

Lemma 5.12 For all $\gamma_\mathrm{A} \in [\![\mathsf{SP}^0_\mathrm{A}]\!]$ and $\gamma_\mathrm{B} \in [\![\mathsf{SP}^0_\mathrm{B}]\!](\gamma_\mathrm{A})$:

(i) If $X \in [\![\mathsf{Set}]\!]$ and $Y(x) \in [\![\mathsf{Set}]\!]$ for each $x \in X$, then $[\![\mathsf{Arg}^0_\mathrm{A}]\!](\gamma_\mathrm{A}, X, Y) \in [\![\mathsf{Set}]\!]$.

(ii) If $X \in [\![\mathsf{Set}]\!]$ and $Y(x) \in [\![\mathsf{Set}]\!]$ for each $x \in X$, $[\![\mathsf{Arg}^0_\mathrm{B}]\!](\gamma_\mathrm{A}, X, Y, \mathsf{intro}_X, \gamma_\mathrm{B}) \in [\![\mathsf{Set}]\!]$. □

We then iterate, using A^α and B^α, in order to reach a fixed point. This uses that fact that both $[\![\mathsf{Arg}^0_\mathrm{A}]\!]$ and $[\![\mathsf{Arg}^0_\mathrm{B}]\!]$ are κ-continuous for large enough κ:

Lemma 5.13

(i) For $\alpha < \mathrm{i}_0$, $A^\alpha \in [\![\mathsf{Set}]\!]$ and $B^\alpha : A^\alpha \to [\![\mathsf{Set}]\!]$.

(ii) For $\alpha < \beta$, $A^\alpha \subseteq A^\beta$ and $B^\alpha(a) \subseteq B^\beta(a)$ for all $a \in A^\alpha$.

(iii) There is $\kappa < \mathrm{i}_0$ such that for all $\alpha \geq \kappa$, $A^\alpha = A^\kappa$ and $B^\alpha(a) = B^\kappa(a)$ for all $a \in A^\alpha$. □

Now we are done, since $[\![A_{\gamma_A,\gamma_B}]\!] = A^{i_0} = A^{\kappa} \in [\![\mathrm{Set}]\!]$, and similarly for $[\![B_{\gamma_A,\gamma_B}]\!]$. We have proved:

Theorem 5.14 There exists a model of the theory of inductive-inductive definitions that can be constructed using ZFC and the existence of two inaccessible cardinals. □

5.2 Container semantics: an extensional normal form

Our axiomatisation of inductive-inductive definitions is based on the idea that inductive types are given by their constructors, which in turn are given by a list of arguments (the codes in SP_A and SP_B). Thus, we are lead to consider a theory which is quite syntactic in nature. Furthermore, since lists (the codes) are inductively defined, it is necessary to use recursion when proving properties about them or constructing functions on them, as we have seen many times already in Chapter 3. This sometimes complicates an intuitively clear idea.

Containers [Abbott, 2003; Abbott et al., 2003, 2005] provide a more semantic notion of data types, where the main mental notion is that of *shapes* and *positions*: each value of a data type has a certain shape, and for each shape, there is a set of positions where data is stored. For instance, a list $\ell : \mathsf{List}(A)$ is completely determined by its length n (the shape of ℓ) and the function $f : \mathsf{Fin}(n) \to A$ which maps each $m : \mathsf{Fin}(n)$ to the value at position m in ℓ. Thus the set of positions for a list of length n is $\mathsf{Fin}(n)$, and we can see (n, f) as a semantic representation of ℓ. On the other hand, each such pair (n, f) determines a list, and we have a bijection (actually isomorphism) $\mathsf{List}(A) \cong (\Sigma n : \mathbb{N})(\mathsf{Fin}(n) \to A)$.

A container $S \lhd P$ consists of a family (S, P), where S : Set are the shapes and $P : S \to \mathrm{Set}$ are the positions. The *extension* of a container $S \lhd P$ is the functor $[\![S \lhd P]\!]_{\mathsf{Cont}} : \mathrm{Set} \to \mathrm{Set}$ defined by $[\![S \lhd P]\!]_{\mathsf{Cont}}(X) = (\Sigma s : S)(P(s) \to X)$. Hence List is the extension of the container $\mathbb{N} \lhd \mathsf{Fin}$, or, in other words, the container $\mathbb{N} \lhd \mathsf{Fin}$ is an induction free representation of the inductive type former List.

We now set out to find a corresponding representation for the functors representing inductive-inductive definitions described in Chapter 4. Containers can be interpreted in any locally Cartesian closed category, so one possible approach would be to prove that the categories involved indeed are locally Cartesian closed. Altenkirch and Morris [2009] follow this approach to construct indexed containers, which represent indexed inductive definitions. However, since we want to represent functors which are not necessarily endofunctors, it is not so clear if this approach would work. Instead, we start with a different observation: every strictly positive inductive type can be represented as the extension of a container, using extensional Type Theory [Dybjer, 1997; Abbott et al., 2004]. Thus, we will look for such a Container Normal Form[1] also for inductive-inductive definitions, and this will tell us what an inductive-inductive container should be.

[1] Thorsten Altenkirch (private communication) once suggested using the abbreviation CNF in order to maximise confusion.

5.2.1 Commuting codes

The idea behind the normal form is quite simple: since noninductive arguments cannot depend on inductive arguments, and A-inductive arguments cannot depend on B-inductive arguments, we can push these arguments to the front, and then combine multiple occurrences into a single occurrence. To show that the meaning of the code is preserved, we need the following isomorphisms, some of them only valid in type theories with function extensionality:

Lemma 5.15 In extensional Type Theory, we have the following isomorphisms:

(i) $(\mathbf{1} \to A) \cong \mathbf{1} \times A \cong A$.

(ii) $(\mathbf{0} \to A) \cong \mathbf{1}$.

(iii) $(\Sigma x : A)(\Sigma y : B(x))C(x,y) \cong (\Sigma p : (\Sigma x : A)B(x))C(\mathsf{fst}(p), \mathsf{snd}(p))$

(iv) $(x : A) \to (y : B(x)) \to C(x,y) \cong (z : (\Sigma x : A)B(x)) \to C(\mathsf{fst}(z), \mathsf{snd}(z))$

(v) $(x : A) \to \big((\Sigma y : B(x))C(x,y)\big) \cong (\Sigma f : (x : A) \to B(x))\big((x : A) \to C(x, f(x))\big)$ □

5.2.1.1 Commuting codes in SP_A

We start with the codes in SP_A.

Lemma 5.16 SP_A is functorial, i.e. if $f : X_\mathrm{ref} \to X'_\mathrm{ref}$, then there is a map $\mathsf{SP}_\mathrm{A}(f) : \mathsf{SP}_\mathrm{A}(X_\mathrm{ref}) \to \mathsf{SP}_\mathrm{A}(X'_\mathrm{ref})$, which lifts to a map

$$\mathsf{SP}_{\mathsf{Arg}_\mathrm{A}}(\gamma, f) : \mathsf{Arg}_\mathrm{A}(X_\mathrm{ref}, \gamma, X, Y, \mathsf{rep}_\mathrm{X} \circ f) \to \mathsf{Arg}_\mathrm{A}(X'_\mathrm{ref}, \mathsf{SP}_\mathrm{A}(f, \gamma), X, Y, \mathsf{rep}_\mathrm{X}) \ .$$

Proof. The map $\mathsf{SP}_\mathrm{A}(f)$ is straightforward to define:

$$\begin{aligned}
\mathsf{SP}_\mathrm{A}(f, \mathsf{nil}) &= \mathsf{nil} \\
\mathsf{SP}_\mathrm{A}(f, \mathsf{non\text{-}ind}(K, \gamma)) &= \mathsf{non\text{-}ind}(K, \lambda k.\, \mathsf{SP}_\mathrm{A}(f, \gamma(k))) \\
\mathsf{SP}_\mathrm{A}(f, \mathsf{A\text{-}ind}(K\, \gamma)) &= \mathsf{A\text{-}ind}(K, \mathsf{SP}_\mathrm{A}(f + \mathsf{id}, \gamma)) \\
\mathsf{SP}_\mathrm{A}(f, \mathsf{B\text{-}ind}(K, h_\mathrm{index}, \gamma)) &= \mathsf{B\text{-}ind}(K, f \circ h_\mathrm{index}, \mathsf{SP}_\mathrm{A}(f, \gamma))
\end{aligned}$$

The only interesting point is perhaps how we have to make a recursive call with $f + \mathsf{id} = [\mathsf{inl} \circ f, \mathsf{inr}] : (X_\mathrm{ref} + K) \to (X'_\mathrm{ref} + K)$ in the A-ind case.

The map $\mathsf{SP}_{\mathsf{Arg}_\mathrm{A}}(\gamma, f)$ is extensionally the identity function, but can be defined also in intensional Type Theory. □

In particular, if f is an isomorphism, so is $\mathsf{SP}_{\mathsf{Arg}_\mathrm{A}}(\gamma, f)$. We will use $\mathsf{SP}_\mathrm{A}(f)$ as a glue when we have codes φ, φ' which are "the same", except that their sets of referable elements X_ref, X'_ref are not equal on the nose, but only isomorphic.

Definition 5.17 We call two codes $\gamma, \phi : \mathsf{SP}^0_{\mathrm{A}}$ *equivalent*, and we write $\gamma \simeq \phi$, if they decode to (naturally) isomorphic sets, i.e.

$$\mathsf{Arg}^0_{\mathrm{A}}(\gamma, X, Y) \cong \mathsf{Arg}^0_{\mathrm{A}}(\phi, X, Y)$$

naturally in (X, Y). ■

The relation $\simeq$ can be naturally extended to open contexts, i.e. codes $\gamma, \phi : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})$, as well. It is obviously an equivalence relation, and we can substitute "equals" for "equals": if $\gamma \simeq \phi$, then e.g. $\mathsf{A\text{-}ind}(K, \gamma) \simeq \mathsf{A\text{-}ind}(K, \phi)$.

Lemma 5.18 The code non-ind can be pushed to the front, and B-ind to the back:

(i) For all $K : \mathsf{Set}$, $S : \mathsf{Set}$ and $\varphi : S \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K)$,

$$\mathsf{A\text{-}ind}(K, \mathsf{non\text{-}ind}(S, \varphi)) \simeq \mathsf{non\text{-}ind}(S, \lambda s.\, \mathsf{A\text{-}ind}(K, \varphi(s)))$$

(ii) For all $K : \mathsf{Set}$, $h_{\mathrm{index}} : K \to X_{\mathrm{ref}}$, $S : \mathsf{Set}$ and $\varphi : S \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})$,

$$\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \mathsf{non\text{-}ind}(S, \varphi)) \simeq \mathsf{non\text{-}ind}(S, \lambda s.\, \mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \varphi(s)))$$

(iii) For all $K_1 : \mathsf{Set}$, $h_{\mathrm{index}} : K_1 \to X_{\mathrm{ref}}$, $K_2 : \mathsf{Set}$ and $\varphi : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K_2)$,

$$\mathsf{B\text{-}ind}(K_1, h_{\mathrm{index}}, \mathsf{A\text{-}ind}(K_2, \varphi)) \simeq \mathsf{A\text{-}ind}(K_2, \mathsf{B\text{-}ind}(K_1, \mathsf{inl} \circ h_{\mathrm{index}}, \varphi))$$

Multiple occurrences of the same code can be combined:

(iv) For all $K_1 : \mathsf{Set}$, $K_2 : K_1 \to \mathsf{Set}$ and $\gamma : (x : K_1) \to K_2(x) \to \mathsf{Set}$,

$$\mathsf{non\text{-}ind}(K_1, \lambda x.\, \mathsf{non\text{-}ind}(K_2(x), \lambda y.\, \gamma(x, y))) \simeq \mathsf{non\text{-}ind}((\Sigma x : K_1) K_2(x), \lambda p.\, \gamma(\mathsf{fst}(p), \mathsf{snd}(p)))$$

(v) For all $K_1 : \mathsf{Set}$, $K_2 : \mathsf{Set}$ and $\varphi : \mathsf{SP}_{\mathrm{A}}((X_{\mathrm{ref}} + K_1) + K_2)$,

$$\mathsf{A\text{-}ind}(K_1, \mathsf{A\text{-}ind}(K_2, \varphi)) \simeq \mathsf{A\text{-}ind}(K_1 + K_2, \mathsf{SP}_{\mathrm{A}}(\alpha, \varphi)) \ ,$$

where $\alpha : (X_{\mathrm{ref}} + K_1) + K_2 \to X_{\mathrm{ref}} + (K_1 + K_2)$ is the isomorphism witnessing the associativity of $+$.

(vi) For all $K_1 : \mathsf{Set}$, $K_2 : \mathsf{Set}$, $h_1 : K_1 \to X_{\mathrm{ref}}$, $h_2 : K_2 \to X_{\mathrm{ref}}$ and $\gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})$,

$$\mathsf{B\text{-}ind}(K_1, h_1, \mathsf{B\text{-}ind}(K_2, h_2, \varphi)) \simeq \mathsf{B\text{-}ind}(K_1 + K_2, [h_1, h_2], \varphi)$$

We have the following base case:

(vii) $\mathsf{nil} \simeq \mathsf{non\text{-}ind}(\mathbf{1}, \mathsf{A\text{-}ind}(\mathbf{0}, \mathsf{B\text{-}ind}(\mathbf{0}, !_{\mathbf{0}}, \mathsf{nil})))$ □

Corollary 5.19 Each code $\gamma : \mathsf{SP}^0_{\mathrm{A}}$ is equivalent to a code of the form

$$\mathsf{non\text{-}ind}(S, \lambda s.\, \mathsf{A\text{-}ind}(P_{\mathrm{A}}(s), \mathsf{B\text{-}ind}(P_{\mathrm{B}}(s), \mathsf{inr} \circ h_{\mathrm{index}}(s), \mathsf{nil}))) \qquad (5.1)$$

where $S : \mathsf{Set}$, $P_{\mathrm{A}} : S \to \mathsf{Set}$, $P_{\mathrm{B}} : S \to \mathsf{Set}$ and $h_{\mathrm{index}} : (s : S) \to P_{\mathrm{B}}(s) \to P_{\mathrm{A}}(s)$.

Proof. Given a code γ, start by replacing all subcodes nil in γ using Lemma 5.18(vii). Now push $\mathsf{non\text{-}ind}$ to the front and $\mathsf{B\text{-}ind}$ to the back using (i) to (iii), combining codes of the same type using (iv) to (vi) as we go along. □

Remark 5.20 Inductive definitions can be seen as a special case of inductive-inductive definitions where the second family $B : A \to \mathrm{Set}$ is arbitrary and the code for first the first set does not use any $\mathsf{B\text{-}ind}$ codes (see Section 3.2.4.1). Hence, applying Corollary 5.19 to such a code, we will end up with $P_\mathrm{B}(s) = 0$ for all $s : S$. Equivalently, we get a code of the form

$$\mathsf{non\text{-}ind}(S, \lambda s.\, \mathsf{A\text{-}ind}(P_\mathrm{A}(s), \mathsf{nil}))\ .$$

But this is the code for the W-type $\mathsf{W}(s : S)P_\mathrm{A}(s)$ from Section 3.2.4.1. Hence we recover Dybjer's [1997] result that inductive definitions can be reduced to W-types in extensional Type Theory.

5.2.1.2 Commuting codes in SP_B

We now repeat the exercise for codes in SP^0_B. We would like to prove that also SP_B is functorial, but first, we need to know that $\mathsf{A\text{-}Term}$ is:

Lemma 5.21 $\mathsf{A\text{-}Term}$ is functorial, i.e. if $f : X_\mathrm{ref} \to X'_\mathrm{ref}$ and $g : Y_\mathrm{ref} \to Y'_\mathrm{ref}$, then there is a function $\mathsf{A\text{-}Term}(\gamma_\mathrm{A}, f, g) : \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref}) \to \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, X'_\mathrm{ref}, Y'_\mathrm{ref})$. Furthermore,

$$\begin{aligned}&\overline{\mathsf{rep}_\mathrm{A}}(\gamma_\mathrm{A}, \mathsf{intro}_\mathrm{A}, \mathsf{rep}_\mathrm{X} \circ f, \mathsf{rep}_\mathrm{index} \circ g, \mathsf{rep}_\mathrm{Y} \circ g, x) = \\ &\qquad\qquad \overline{\mathsf{rep}_\mathrm{A}}(\gamma_\mathrm{A}, \mathsf{intro}_\mathrm{A}, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y}, \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, f, g)(x))\end{aligned}$$

for all $x : \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref})$. □

Proof. This follows immediately from the functoriality of $\mathsf{Arg}^0_\mathrm{A}$, if we simultaneously prove that also $\mathsf{B\text{-}Term}$ is functorial, i.e. that given $f : X_\mathrm{ref} \to X'_\mathrm{ref}$ and $g : Y_\mathrm{ref} \to Y'_\mathrm{ref}$, there is a map

$$\begin{aligned}&\mathsf{B\text{-}Term}(\gamma_\mathrm{A}, f, g)(x) : \mathsf{B\text{-}Term}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref}, x) \\ &\qquad\qquad \to \mathsf{B\text{-}Term}(\gamma_\mathrm{A}, X'_\mathrm{ref}, Y'_\mathrm{ref}, \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, f, g)(x))\end{aligned}$$

for each $x : \mathsf{A\text{-}Term}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref})$. □

Lemma 5.22 SP_B is functorial, i.e. if $f : X_\mathrm{ref} \to X'_\mathrm{ref}$ and $g : Y_\mathrm{ref} \to Y'_\mathrm{ref}$, then there is a map $\mathsf{SP}_\mathrm{B}(f, g) : \mathsf{SP}_\mathrm{B}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref}) \to \mathsf{SP}_\mathrm{B}(\gamma_\mathrm{A}, X'_\mathrm{ref}, Y'_\mathrm{ref})$, which lifts to a map

$$\begin{aligned}&\mathsf{SP}_{\mathsf{Arg}_\mathrm{B}}(\gamma, f, g) : \mathsf{Arg}_\mathrm{B}(\gamma_\mathrm{A}, X_\mathrm{ref}, Y_\mathrm{ref}, \gamma, X, Y, \mathsf{rep}_\mathrm{X} \circ f, \mathsf{rep}_\mathrm{index} \circ g, \mathsf{rep}_\mathrm{Y} \circ g) \\ &\qquad\qquad \to \mathsf{Arg}_\mathrm{B}(\gamma_\mathrm{A}, X'_\mathrm{ref}, Y'_\mathrm{ref}, \mathsf{SP}_\mathrm{B}(f, g, \gamma), X, Y, \mathsf{rep}_\mathrm{X}, \mathsf{rep}_\mathrm{index}, \mathsf{rep}_\mathrm{Y})\ .\end{aligned}$$

Proof. The map $\mathsf{SP}_{\mathrm{B}}(f,g)$ is defined like $\mathsf{SP}_{\mathrm{A}}(f)$:

$$\begin{aligned}
\mathsf{SP}_{\mathrm{B}}(f,g,\mathsf{nil}(a)) &= \mathsf{nil}(\mathsf{A\text{-}Term}(\gamma_{\mathrm{A}},f,g)(a)) \\
\mathsf{SP}_{\mathrm{B}}(f,g,\mathsf{non\text{-}ind}(K,\gamma)) &= \mathsf{non\text{-}ind}(K,\lambda k.\,\mathsf{SP}_{\mathrm{B}}(f,g,\gamma(k))) \\
\mathsf{SP}_{\mathrm{B}}(f,g,\mathsf{A\text{-}ind}(K\,\gamma)) &= \mathsf{A\text{-}ind}(K,\mathsf{SP}_{\mathrm{B}}(f+\mathsf{id},g,\gamma)) \\
\mathsf{SP}_{\mathrm{B}}(f,g,\mathsf{B\text{-}ind}(K,h_{\mathrm{index}},\gamma)) &= \mathsf{B\text{-}ind}(K,\mathsf{A\text{-}Term}(\gamma_{\mathrm{A}},f,g)\circ h_{\mathrm{index}},\mathsf{SP}_{\mathrm{B}}(f,g+\mathsf{id},\gamma))\,\square
\end{aligned}$$

Definition 5.23 We call two codes $\gamma,\phi : \mathsf{SP}^{0}_{\mathrm{B}}(\gamma_{\mathrm{A}})$ *equivalent*, written $\gamma \simeq \phi$, if

(i) they decode to (naturally) isomorphic sets, i.e. there is an isomorphism

$$f_{X,Y,\mathsf{intro}_{\mathrm{A}}} : \mathsf{Arg}^{0}_{\mathrm{B}}(\gamma,X,Y,\mathsf{intro}_{\mathrm{A}}) \cong \mathsf{Arg}^{0}_{\mathrm{A}}(\phi,X,Y,\mathsf{intro}_{\mathrm{A}})$$

natural in $(X,Y,\mathsf{intro}_{\mathrm{A}})$, and

(ii) they target the same index, i.e.

$$\mathsf{Index}^{0}_{\mathrm{B}}(\gamma,x) = \mathsf{Index}^{0}_{\mathrm{B}}(\phi,f_{X,Y,\mathsf{intro}_{\mathrm{A}}}(x))$$

for all $x : \mathsf{Arg}^{0}_{\mathrm{B}}(\gamma,X,Y,\mathsf{intro}_{\mathrm{A}})$. ▣

Lemma 5.24 The code non-ind can be pushed to the front, and B-ind to the back:

(i) For all $K : \mathsf{Set}$, $S : \mathsf{Set}$ and $\varphi : S \to \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}+K,Y_{\mathrm{ref}},\gamma_{\mathrm{A}})$,

$$\mathsf{A\text{-}ind}(K,\mathsf{non\text{-}ind}(S,\varphi)) \simeq \mathsf{non\text{-}ind}(S,\lambda s.\,\mathsf{A\text{-}ind}(K,\varphi(s)))$$

(ii) For all $K,S : \mathsf{Set}$, $h : K \to \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}},X_{\mathrm{ref}},Y_{\mathrm{ref}})$ and $\varphi : S \to \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}},Y_{\mathrm{ref}}+K,\gamma_{\mathrm{A}})$,

$$\mathsf{B\text{-}ind}(K,h,\mathsf{non\text{-}ind}(S,\varphi)) \simeq \mathsf{non\text{-}ind}(S,\lambda s.\,\mathsf{B\text{-}ind}(K,h,\varphi(s)))$$

(iii) For all $K_1 : \mathsf{Set}$, $h_{\mathrm{index}} : K_1 \to \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}},X_{\mathrm{ref}},Y_{\mathrm{ref}})$, $K_2 : \mathsf{Set}$ and $\varphi : \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}}+K_2,Y_{\mathrm{ref}}+K_1,\gamma_{\mathrm{A}})$,

$$\mathsf{B\text{-}ind}(K_1,h_{\mathrm{index}},\mathsf{A\text{-}ind}(K_2,\varphi)) \simeq \mathsf{A\text{-}ind}(K_2,\mathsf{B\text{-}ind}(K_1,\mathsf{A\text{-}Term}(\gamma_{\mathrm{A}},\mathsf{inl},\mathsf{id})\circ h_{\mathrm{index}},\varphi))$$

Multiple occurrences of non-ind and A-ind can be combined:

(iv) For all $K_1 : \mathsf{Set}$, $K_2 : K_1 \to \mathsf{Set}$ and $\gamma : (x : K_1) \to K_2(x) \to \mathsf{SP}_{\mathrm{B}}(X_{\mathrm{ref}},Y_{\mathrm{ref}},\gamma_{\mathrm{A}})$,

$$\begin{aligned}
&\mathsf{non\text{-}ind}(K_1,\lambda x.\,\mathsf{non\text{-}ind}(K_2(x),\lambda y.\,\gamma(x,y))) \simeq \\
&\qquad\qquad \mathsf{non\text{-}ind}((\Sigma x : K_1)K_2(x),\lambda p.\,\gamma(\mathsf{fst}(p),\mathsf{snd}(p)))
\end{aligned}$$

(v) For all $K_1 : \mathsf{Set}$, $K_2 : \mathsf{Set}$ and $\varphi : \mathsf{SP}_{\mathrm{B}}((X_{\mathrm{ref}}+K_1)+K_2,Y_{\mathrm{ref}},\gamma_{\mathrm{A}})$,

$$\mathsf{A\text{-}ind}(K_1,\mathsf{A\text{-}ind}(K_2,\varphi)) \simeq \mathsf{A\text{-}ind}(K_1+K_2,\mathsf{SP}_{\mathrm{B}}(\alpha,\mathsf{id},\varphi)) \ ,$$

where $\alpha : (X_{\mathrm{ref}}+K_1)+K_2 \to X_{\mathrm{ref}}+(K_1+K_2)$ is the isomorphism witnessing the associativity of $+$.

We have the following base case:

(vi) $\mathsf{nil}(a) \simeq \mathsf{non\text{-}ind}(\mathbf{1}, \mathsf{A\text{-}ind}(\mathbf{0}, \mathsf{B\text{-}ind}(\mathbf{0}, !_{\mathbf{0}}, \mathsf{nil}(\mathsf{A\text{-}Term}(\mathsf{inl}, \mathsf{inl}, a))))))$ □

Note how the combining of multiple B-ind codes into one is missing from the lemma – later arguments might now depend on earlier ones by including them in their index via the constructor $\mathsf{intro}_{\mathrm{A}}$. For instance, let $\gamma_{\mathrm{A}} = \mathsf{A\text{-}ind}(\mathbf{1}, \mathsf{B\text{-}ind}(\mathbf{1}, \mathsf{inr}, \mathsf{nil}))$. In other words, γ_{A} represents a data type A with constructor $\mathsf{intro}_{\mathrm{A}} : (a : A) \to B(a) \to A$ for some as of yet unspecified data type $B : A \to \mathsf{Set}$. Now consider e.g. the code

$$\gamma_{\mathrm{B}} = \mathsf{A\text{-}ind}_{\mathbf{1}}(\mathsf{B\text{-}ind}_{\mathbf{1}}(\mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star)), \mathsf{B\text{-}ind}_{\mathbf{1}}(\mathsf{arg}(\mathsf{b}_{\mathrm{ref}}(\mathsf{inr}(\star)), \star, \star), \mathsf{nil}(\mathsf{a}_{\mathrm{ref}}(\mathsf{inr}(\star))))))$$

which represents a constructor

$$\mathsf{intro}_{\mathrm{B}} : (a : A) \to (b : B(a)) \to B(\mathsf{intro}_{\mathrm{A}}(a, b)) \to B(a) \enspace .$$

We need the *argument* $b : B(a)$ before we can describe the *index* $\mathsf{intro}_{\mathrm{A}}(a, b)$, hence these two arguments cannot be combined. The best we can do when it comes to normal forms is the following:

Corollary 5.25 Each code $\gamma_{\mathrm{B}} : \mathsf{SP}^{0}_{\mathrm{B}}(\gamma_{\mathrm{A}})$ is equivalent to a code of the form

$$\begin{aligned} &\mathsf{non\text{-}ind}(S, \lambda s.\, \mathsf{A\text{-}ind}(P_{\mathrm{A}}(s), \\ &\qquad \underbrace{\mathsf{B\text{-}ind}(P_{\mathrm{B},0}(s), h_0(s), \ldots \mathsf{B\text{-}ind}(P_{\mathrm{B},n(s)}(s), h_{n(s)}(s), \mathsf{nil}(a(s))))))}_{n(s)\text{ many}} \end{aligned} \tag{5.2}$$

where $S : \mathsf{Set}$, $P_{\mathrm{A}} : S \to \mathsf{Set}$, $n : S \to \mathbb{N}$,

$$\begin{aligned} P_{\mathrm{B},i} &: S \to \mathsf{Set} \\ h_i &: (s : S) \to P_{\mathrm{B},i}(s) \to \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, P_{\mathrm{A}}(s), P_{\mathrm{B},0}(s) + \ldots P_{\mathrm{B},i-1}(s)) \end{aligned}$$

for $0 \leq i \leq n(s)$ and $a : (s : S) \to \mathsf{A\text{-}Term}(\gamma_{\mathrm{A}}, P_{\mathrm{A}}(s), P_{\mathrm{B},0}(s) + \ldots P_{\mathrm{B},n(s)}(s))$.

Proof. As before, push non-ind to the front and B-ind to the end. As multiple occurrences all codes but B-ind can be combined, we will end up with a code as described above. □

5.2.2 Inductive-inductive containers

We now reap the benefits of the work of the last section by reading off the definition of an inductive-inductive container from Corollaries 5.19 and 5.25.

Definition 5.26 (inductive-inductive container) An *inductive-inductive container* (C_A, C_B) is given by the following data, and decoded as follows:

- $C_A = (S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{B}}, h^A_{\mathrm{index}})$, where
  - $S^A : \mathsf{Set}$,

- $P^A_{\mathrm{A}} : S^A \to \mathsf{Set}$,
- $P^A_{\mathrm{B}} : S^A \to \mathsf{Set}$, and
- $h^A_{\mathrm{index}} : (s : S^A) \to P_{\mathrm{B}}(s) \to P_{\mathrm{A}}(s)$.

- The *extension* of $C_A = (S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{B}}, h^A_{\mathrm{index}})$ is the functor $[\![C_A]\!] : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ defined by

$$[\![(S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{A}}, h^A_{\mathrm{index}})]\!](X,Y) = \\ (\Sigma s : S^A)(\Pi f : P^A_{\mathrm{A}}(s) \to X)(\Pi x : P^A_{\mathrm{B}}(s))Y(f(h^A_{\mathrm{index}}(s,x))) \ .$$

This extends to an action on morphisms in the obvious way.

- Given $C_A = (S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{B}}, h^A_{\mathrm{index}})$ and sets X_{ref}, Y_{ref}, the set $\mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ is inductively generated by the constructors

$$\begin{aligned}
&\mathsf{a}_{\mathrm{ref}} : X_{\mathrm{ref}} \to \mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}})\\
&\mathsf{b}_{\mathrm{ref}} : Y_{\mathrm{ref}} \to \mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}})\\
&\mathsf{arg} : [\![C_A]\!](\mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}), \mathsf{B\text{-}Term}_{C_A}) \to \mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}})
\end{aligned}$$

where $\mathsf{B\text{-}Term}_{C_A}(\mathsf{a}_{\mathrm{ref}}(x)) = \mathsf{B\text{-}Term}_{C_A}(\mathsf{arg}(x)) = \mathbf{0}$ and $\mathsf{B\text{-}Term}_{C_A}(\mathsf{b}_{\mathrm{ref}}(x)) = \mathbf{1}$.

Given $\mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X$, $\mathsf{rep}_{\mathrm{index}} : Y_{\mathrm{ref}} \to X$ and $\mathsf{rep}_{\mathrm{Y}} : (y : Y_{\mathrm{ref}}) \to Y(\mathsf{rep}_{\mathrm{index}}(y))$, the functions

$$\begin{aligned}
&\overline{\mathsf{rep}}_{\mathrm{A}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}) : \mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}) \to X\\
&\overline{\mathsf{rep}}_{\mathrm{B}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}) : (x : \mathsf{A\text{-}Term}_{C_A}(X_{\mathrm{ref}}, Y_{\mathrm{ref}})) \to Y(\overline{\mathsf{rep}}_{\mathrm{A}C_A}(\ldots, x))
\end{aligned}$$

are defined by

$$\begin{aligned}
&\overline{\mathsf{rep}}_{\mathrm{A}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{a}_{\mathrm{ref}}(x)) = \mathsf{rep}_{\mathrm{X}}(x)\\
&\overline{\mathsf{rep}}_{\mathrm{A}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{b}_{\mathrm{ref}}(x)) = \mathsf{rep}_{\mathrm{index}}(x)\\
&\overline{\mathsf{rep}}_{\mathrm{A}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{arg}(x)) = [\![C_A]\!](\overline{\mathsf{rep}}_{\mathrm{A}C_A}(\ldots), \overline{\mathsf{rep}}_{\mathrm{B}C_A}(\ldots), x)
\end{aligned}$$

$$\begin{aligned}
&\overline{\mathsf{rep}}_{\mathrm{B}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{a}_{\mathrm{ref}}(x), y) = !(y)\\
&\overline{\mathsf{rep}}_{\mathrm{B}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{b}_{\mathrm{ref}}(x), \star) = \mathsf{rep}_{\mathrm{Y}}(x)\\
&\overline{\mathsf{rep}}_{\mathrm{B}C_A}(\mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \mathsf{arg}(x), y) = !(y)
\end{aligned}$$

- $C_B = (S^B, P^B_{\mathrm{A}}, n^B, P^B_{\mathrm{B}}, h^B, a^B)$, where
  - $S^B : \mathsf{Set}$,
  - $P^B_{\mathrm{A}} : S \to \mathsf{Set}$,
  - $n^B : S \to \mathbb{N}$,

- $P^B_{\mathrm{B}} : (s : S) \to \mathsf{Fin}(n(s)) \to \mathsf{Set}$,
- $h^B : (s : S) \to (i : \mathsf{Fin}(n(s))) \to P^B_{\mathrm{B}}(s,i)$
  $\to \mathsf{A\text{-}Term}_{C_A}(P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s,0) + \ldots + P^B_{\mathrm{B}}(s,i-1))$, and
- $a^B : (s : S) \to \mathsf{A\text{-}Term}_{C_A}(P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s,0) + \ldots + P^B_{\mathrm{B}}(s,n(s)))$,

The extension of the inductive-inductive container (C_A, C_B) is a functor

$$\llbracket C_A, C_B \rrbracket : \mathsf{Dialg}(\llbracket C_A \rrbracket, U) \to \mathsf{Fam}(\mathsf{Set}) \ ,$$

where $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ is the index set functor $U(X,Y) = X$. The functor $\llbracket C_A, C_B \rrbracket$ is defined by

$$\llbracket C_A, C_B \rrbracket(X, Y, \mathsf{intro}_{\mathrm{A}}) = (\llbracket C_A \rrbracket(X,Y), \llbracket C_B \rrbracket(X, Y, \mathsf{intro}_{\mathrm{A}}))$$

where $\llbracket C_B \rrbracket : \llbracket C_A \rrbracket \to \mathsf{Set}$ is defined by

$$\begin{aligned}
&\llbracket (S^B, P^B_{\mathrm{A}}, n^B, P^B_{\mathrm{B}}, h^B, a^B) \rrbracket (X, Y, \mathsf{intro}_{\mathrm{A}})(z) \\
&\quad = (\Sigma s : S^B)\big((\Pi f : P^B_{\mathrm{A}}(s) \to X) \\
&\qquad\qquad (\Pi g_0 : (\Pi x : P^B_{\mathrm{B}}(s,0)) Y(\overline{h}(s,0,x))) \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad (\Pi g_{n(s)} : (\Pi x : P^B_{\mathrm{B}}(s,n(s))) Y(\overline{h}(s,n(s),x)))) \\
&\qquad\qquad\quad \overline{\mathsf{rep}_{\mathrm{A}}}_{C_A}(f, [\overline{h}(s,0), \ldots, \overline{h}(s,i)], [g_0, \ldots, g_i], a(s)) \equiv_X \mathsf{intro}_{\mathrm{A}}(z)\big)
\end{aligned}$$

where

$$\begin{aligned}
\overline{h}(s,0) &= \overline{\mathsf{rep}_{\mathrm{A}}}_{C_A}(f, !, !) \circ h^b(s,0) \\
\overline{h}(s,i+1) &= \overline{\mathsf{rep}_{\mathrm{A}}}_{C_A}(f, [\overline{h}(s,0), \ldots, \overline{h}(s,i)], [g_0, \ldots, g_i]) \circ h^b(s,i+1) \ . \qquad \square
\end{aligned}$$

By construction, Corollaries 5.19 and 5.25 together now say that each inductive-inductive definition can be interpreted as an inductive-inductive container.

The "number" $n : S \to \mathbb{N}$ is reminiscent of n-ary containers $S \lhd P$, where P has type $P : S \to \mathsf{Fin}(n) \to \mathsf{Set}$ [Abbott et al., 2003], except that our n is allowed to vary with the shape. Of course, if the number of shapes is finite, we can just choose a uniform $n' = \max_{s:S}(n(s))$ and pad out the extra positions with empty ones:

$$P'_{\mathrm{B}}(s,i) = \begin{cases} P_{\mathrm{B}}(s,i) & \text{if } i \leq n(s) \\ \mathbf{0} & \text{otherwise} \end{cases} \qquad\qquad h'(s,i) = \begin{cases} h(s,i) & \text{if } i \leq n(s) \\ ! & \text{otherwise} \end{cases}$$

We then have that $\llbracket C_A, (S, P_{\mathrm{A}}, n, P_{\mathrm{B}}, h, i) \rrbracket \cong \llbracket C_A, (S, P_{\mathrm{A}}, n', P'_{\mathrm{B}}, h', i) \rrbracket$. However, in general, $n : S \to \mathbb{N}$ can be unbounded.

Example 5.27 (Context and types as an inductive-inductive container) We express the contexts and types from Example 3.1 as an inductive-inductive container $(C_{\mathsf{Ctxt}}, C_{\mathsf{Ty}})$.

Let us start with $C_{\mathsf{Ctxt}} = (S^{\mathsf{Ctxt}}, P^{\mathsf{Ctxt}}_{\mathrm{A}}, P^{\mathsf{Ctxt}}_{\mathrm{B}}, h^{\mathsf{Ctxt}}_{\mathrm{index}})$. The data type Ctxt has two constructors, thus we set $S^{\mathsf{Ctxt}} = \mathbf{2}$. The first constructor ε has no inductive arguments at all, thus we choose $P^{\mathsf{Ctxt}}_{\mathrm{A}}(\mathsf{ff}) = P^{\mathsf{Ctxt}}_{\mathrm{B}}(\mathsf{ff}) = \mathbf{0}$ with $h^{\mathsf{Ctxt}}_{\mathrm{index}}(\mathsf{ff}, x) = !(x)$. The second constructor

$$\rhd : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Ctxt}$$

has one Ctxt -inductive and one Ty-inductive argument respectively, thus we choose $P^{\mathsf{Ctxt}}_{\mathrm{A}}(\mathsf{tt}) = P^{\mathsf{Ctxt}}_{\mathrm{B}}(\mathsf{tt}) = \mathbf{1}$ with $h^{\mathsf{Ctxt}}_{\mathrm{index}}(\mathsf{tt}, \star) = \star$. This concludes the definition of $C_{\mathsf{Ctxt}} = (S^{\mathsf{Ctxt}}, P^{\mathsf{Ctxt}}_{\mathrm{A}}, P^{\mathsf{Ctxt}}_{\mathrm{B}}, h^{\mathsf{Ctxt}}_{\mathrm{index}})$.

We now move on to $C_{\mathsf{Ty}} = (S^{\mathsf{Ty}}, P^{\mathsf{Ty}}_{\mathrm{A}}, n^{\mathsf{Ty}}, P^{\mathsf{Ty}}_{\mathrm{B}}, h^{\mathsf{Ty}}, a^{\mathsf{Ty}})$. Since also Ty has two constructors, we let $S^{\mathsf{Ty}} = \mathbf{2}$ as well. The first constructor $\iota : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma)$ has one Ctxt-inductive and no Ty-inductive arguments, so we let $P^{\mathsf{Ty}}_{\mathrm{A}}(\mathsf{ff}) = \mathbf{1}$ and $n^{\mathsf{Ty}}(\mathsf{ff}) = 0$ (with $P^{\mathsf{Ty}}_{\mathrm{B}}(\mathsf{ff})$ and $h^{\mathsf{Ty}}(\mathsf{ff})$ trivially given by ex falso quod libet), and finally, since the index is the only Ctxt-inductive argument, we let $a^{\mathsf{Ty}}(\mathsf{ff}) = \mathsf{a}_{\mathrm{ref}}(\star)$. The second constructor

$$\Pi : (\Gamma : \mathsf{Ctxt}) \to (\sigma : \mathsf{Ty}(\Gamma)) \to \mathsf{Ty}(\Gamma \rhd \sigma) \to \mathsf{Ty}(\Gamma)$$

has one Ctxt-inductive and two Ty-inductive arguments, so we let $P^{\mathsf{Ty}}_{\mathrm{A}}(\mathsf{tt}) = \mathbf{1}$ and $n^{\mathsf{Ty}}(\mathsf{tt}) = 2$ with $P^{\mathsf{Ty}}_{\mathrm{B}}(\mathsf{tt}, 0) = P^{\mathsf{Ty}}_{\mathrm{B}}(\mathsf{tt}, 1) = \mathbf{1}$. The index for the first Ty-inductive argument σ is the Ctxt-inductive argument Γ, so we let $h^{\mathsf{Ty}}(\mathsf{tt}, 0, \star) = \mathsf{a}_{\mathrm{ref}}(\star)$, while the index of the second one should be $\Gamma \rhd \sigma$. Hence we let $h^{\mathsf{Ty}}(\mathsf{tt}, 1, \star) = \mathsf{arg}(\langle \mathsf{tt}, \lambda\_.\, \star, \lambda\_.\, \star \rangle)$ since $\mathsf{arg}(\langle \mathsf{tt}, \lambda\_.\, \star, \lambda\_.\, \star \rangle)$ represents the constructor $\rhd$ applied to the only elements Γ and σ we have access to. Finally, the index targeted by the constructor is Γ, so we again define $a^{\mathsf{Ty}}(\mathsf{tt}) = \mathsf{a}_{\mathrm{ref}}(\star)$. ∎

Remark 5.28 The container literature also defines morphisms between containers, which represent natural transformations between the corresponding functors. Hence containers and container morphisms form a category Cont, and the embedding Cont → $\mathsf{Set}^{\mathsf{Set}}$ is in fact full and faithful. Thus, we can use the locally small category Cont to represent objects in the (even locally) *large* functor category $\mathsf{Set}^{\mathsf{Set}}$. It is not hard to define morphisms also between inductive-inductive containers. It would be interesting to see if there is a corresponding full and faithfulness result also in this case.

The definition of inductive-inductive containers stands on its own; no reference was made to the axiomatisation in Chapter 3. The price we paid for this was repeating the definition of A-Term and $\overline{\mathsf{rep}_{\mathrm{A}}}$, specialised to the normal form code in Corollary 5.19. Even though there are no inductively presented codes present in the definition anymore, the situation is not completely satisfactory, as A-Term is still inductively defined. This means that constructions on inductive-inductive containers still need to use a small amount of recursion.

5.2.3 Graded inductive-inductive containers

We now present a subset of inductive-inductive containers, which we call *graded* inductive-inductive containers, and which are presented in an induction-free way.

This makes them very easy to reason about. I chose the name *graded*, since these inductive-inductive containers correspond to data types where the generalised arguments $g : (x : P_{\mathrm{B}}(s)) \to B(i(x))$ in the constructor for the second set $B : A \to \mathsf{Set}$ can be decomposed into arguments which use respectively $0, 1, 2, \ldots, n$ constructors for the first set A respectively. Not all inductive-inductive containers are of this form, but we will see that all examples we have considered so far are.

The first part C_A of a graded inductive-inductive container is the same as for a general inductive-inductive container. We give the full definition for completeness (the reader with a good memory of general inductive-inductive containers can skip straight to the third bullet point):

Definition 5.29 (Graded inductive-inductive container) A *graded inductive-inductive container* (C_A, C_B) is given by the following data, and decoded the following way:

- $C_A = (S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{B}}, h^A_{\mathrm{index}})$, where
  - $S^A : \mathsf{Set}$,
  - $P^A_{\mathrm{A}} : S \to \mathsf{Set}$,
  - $P^A_{\mathrm{B}} : S \to \mathsf{Set}$, and
  - $h^A_{\mathrm{index}} : (s : S) \to P_{\mathrm{B}}(s) \to P_{\mathrm{A}}(s)$.
- The *extension* of $C_A = (S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{B}}, h^A_{\mathrm{index}})$ is the functor $[\![C_A]\!] : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ defined by

  $$[\![(S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{A}}, h^A_{\mathrm{index}})]\!](X, Y) = \\ (\Sigma s : S^A)(\Pi f : P^A_{\mathrm{A}}(s) \to X)(\Pi x : P^A_{\mathrm{B}}(s))Y(f(h^A_{\mathrm{index}}(s, x))) \ .$$

  This extends to an action on morphisms in the obvious way.
- $C_B = (S^B, P^B_{\mathrm{A}}, n^B, P^B_{\mathrm{B}}, a^B)$, where
  - $S^B : \mathsf{Set}$,
  - $P^B_{\mathrm{A}} : S \to \mathsf{Set}$,
  - $n^B : S \to \mathbb{N}$,
  - $P^B_{\mathrm{B}} : (s : S) \to (i : \mathsf{Fin}(n(s))) \to [\![C_A]\!]^i(P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s, 0), \ldots, P^B_{\mathrm{B}}(i-1)) \to \mathsf{Set}$,
  - $a^B : (s : S) \to \Sigma_{i=0}^{n(s)} [\![C_A]\!](P^B_{\mathrm{A}}(s), \vec{P^B_{\mathrm{B}}}(s))$,

  where $[\![C_A]\!]^0(X) = X$ and

  $$[\![C_A]\!]^{i+1}(X, Y_1, \ldots, Y_{i+1}) = \\ [\![C_A]\!]([\![C_A]\!]^0(X) + \ldots + [\![C_A]\!]^i(X, Y_1, \ldots, Y_i), [Y_1, \ldots, Y_{i+1}]) \ .$$

The extension of the graded inductive-inductive container (C_A, C_B) is a functor

$$[\![C_A, C_B]\!] : \mathsf{Dialg}([\![C_A]\!], U) \to \mathsf{Fam}(\mathsf{Set}) \ ,$$

where $U : \mathsf{Fam}(\mathsf{Set}) \to \mathsf{Set}$ is the index set functor $U(X, Y) = X$. The functor $[\![C_A, C_B]\!]$ is defined by

$$\begin{aligned} &[\![C_A, C_B]\!](X, Y, \mathsf{intro}_\mathsf{A}) = \\ &\qquad ([\![C_A]\!](X, Y), \lambda x.\,(\Sigma y : [\![C_B]\!](X, Y, \mathsf{intro}_\mathsf{A}))[\![C_B]\!]_{\mathsf{Index}_\mathsf{B}}(y) \equiv_X \mathsf{intro}_\mathsf{A}(x)) \end{aligned}$$

where $[\![C_B]\!]$ is defined by

$$\begin{aligned} &[\![(S^B, P^B_\mathrm{A}, n^B, P^B_\mathrm{B}, a^B)]\!](X, Y, \mathsf{intro}_\mathsf{A}) \\ &\ = (\Sigma s : S^B)\big((\Pi f : P^B_\mathrm{A}(s) \to X) \\ &\qquad\qquad (\Pi g_0 : (\Pi x : P^B_\mathrm{A}(s))P^B_\mathrm{B}(s, 0, x) \to Y(f(x))) \\ &\qquad\qquad (\Pi g_1 : (\Pi x : [\![C_A]\!](P^B_\mathrm{A}(s), P^B_\mathrm{B}(s, 0)))P^B_\mathrm{B}(s, 1, x) \to Y(\overline{f}(s, 1, x))) \\ &\qquad\qquad\qquad \vdots \\ &\qquad\qquad (\Pi g_{n(s)} : (\Pi x : [\![C_A]\!]^{n(s)}(P^B_\mathrm{A}(s), P^B_\mathrm{B}(s, 0), \ldots, P^B_\mathrm{B}(s, n(s) - 1))) \\ &\qquad\qquad\quad P^B_\mathrm{B}(s, n(s), x) \to Y(\overline{f}(s, n(s), x)))\big) \end{aligned}$$

and $[\![C_B]\!]_{\mathsf{Index}_\mathsf{B}}(X, Y, \mathsf{intro}_\mathsf{A}) : [\![C_B]\!](X, Y, \mathsf{intro}_\mathsf{A}) \to X$ is defined by

$$[\![C_B]\!]_{\mathsf{Index}_\mathsf{B}}(X, Y, \mathsf{intro}_\mathsf{A})(s, f, g_0, \ldots, g_{n(s)}) = [\overline{f}(s, 0), \ldots, \overline{f}(s, n(s))](a(s))$$

where

$$\begin{aligned} \overline{f}(s, 0) &= f \\ \overline{f}(s, i + 1) &= \mathsf{intro}_\mathsf{A} \circ [\![C_A]\!]([\overline{f}(s, 0), \ldots, \overline{f}(s, i)], [g_0, \ldots, g_i]) \ . \qquad \blacksquare \end{aligned}$$

Example 5.30 (Contexts and types as a graded inductive-inductive container) We recast the contexts and types from Example 5.27 as a graded inductive-inductive container. The first component C_{Ctxt} stays exactly the same. For the second component $C_{\mathsf{Ty}} = (S^{\mathsf{Ty}}, P^{\mathsf{Ty}}_\mathrm{A}, n^{\mathsf{Ty}}, P^{\mathsf{Ty}}_\mathrm{B}, a^{\mathsf{Ty}})$, the shapes, Ctxt-positions and number of Ty-positions are the same as before: the shapes are $S^{\mathsf{Ty}} = \mathbf{2}$, we have $P^{\mathsf{Ty}}_\mathrm{A}(\mathsf{ff}) = P^{\mathsf{Ty}}_\mathrm{A}(\mathsf{tt}) = \mathbf{1}$ and $n^{\mathsf{Ty}}(\mathsf{ff}) = 0$, $n^{\mathsf{Ty}}(\mathsf{tt}) = 2$. The difference compared to Example 5.27 is how we describe the Ty-positions. With a graded container, we describe how many inductive arguments of a given constructor-shape we have:

$$P^{\mathsf{Ty}}_\mathrm{B}(\mathsf{tt}, 0, x) = \mathbf{1} \tag{5.3}$$

$$P^{\mathsf{Ty}}_\mathrm{B}(\mathsf{tt}, 1, \langle \mathsf{ff}, y \rangle) = \mathbf{0} \tag{5.4}$$

$$P^{\mathsf{Ty}}_\mathrm{B}(\mathsf{tt}, 1, \langle \mathsf{tt}, y \rangle) = \mathbf{1} \tag{5.5}$$

Equation (5.3) says that we have one inductive argument not using a constructor, Equation (5.4) that we have no inductive argument targeting ε (which is represented by a tuple of the form $\langle \mathsf{ff}, y\rangle$) and Equation (5.5) that we have one inductive argument targetting the constructor $\rhd$ (which is represented by a tuple of the form $\langle \mathsf{tt}, y\rangle$).

Finally the index of the constructed element should be the only Ctxt-inductive argument in both constructors for Ty, i.e. no constructor involved in the index, so we define $a^{\mathsf{Ty}}(\mathsf{ff}) = \mathsf{inl}(\star)$ and $a^{\mathsf{Ty}}(\mathsf{tt}) = \mathsf{inl}(\star)$. ∎

Not every inductive-inductive container has a gradation. Consider for instance the container which represents the following inductive-inductive definition of (A, B): the set A has two constructors $\mathsf{base} : \mathsf{A}$ and $\mathsf{intro_A} : A \to A$, and B has one constructor

$$\mathsf{intro_B} : (a : A) \to (f : (n : \mathbb{N}) \to B(\mathsf{elim}_{\mathbb{N}}(a, \lambda m.\, \lambda \widetilde{m}.\, \mathsf{intro_A}(\widetilde{m}), n))) \to B(a)$$

In other words, $\mathsf{intro_B}$ has arguments $a : A$ and $f(n) : B(\mathsf{intro}^n_{\mathsf{A}}(a))$ for each $n : \mathbb{N}$. As a container, this data type can be represented by (C_A, C_B) where $C_A = (\mathbf{2}, P^A_{(A)}, \lambda\_.\, \mathbf{0}, !)$ with $P^A_{(A)}(\mathsf{ff}) = \mathbf{0}$ and $P^A_{(A)}(\mathsf{tt}) = \mathbf{1}$, and

$$C_B = (\mathbf{1}, \mathbf{1}, 1, \mathbb{N}, \mathsf{elim}_{\mathbb{N}}(\mathsf{a_{ref}}(\star), \lambda m.\, \lambda \widetilde{m}.\, \mathsf{arg}(\mathsf{tt}, \widetilde{m})), \mathsf{a_{ref}}(\star))$$

i.e. we have one shape, one A-position and a B-position for each natural number n, whose index is $\mathsf{arg}(\mathsf{tt}, \mathsf{arg}(\mathsf{tt}, \mathsf{arg}(\ldots, \mathsf{a_{ref}}(\mathbf{1}))))$ (n occurrences of arg). There is no corresponding graded container, as this would need an infinite number of arguments

$$\begin{aligned} &f(0) : B(a) \\ &f(1) : B(\mathsf{intro_A}(a)) \\ &f(2) : B(\mathsf{intro}^2_{\mathsf{A}}(a)) \\ &\quad\vdots \end{aligned}$$

since a graded container always groups together the arguments that use the same depth of constructors.

Note that this counterexample is only possible because the inductive-inductive definition is degenerate: the constructor for A does not refer to B. For proper inductive-inductive definitions, one could expect a gradation always to exist, since arguments must be introduced in a certain order: $b : B(a)$ before $b' : B(\mathsf{intro_A}(a, b))$, for instance. However, definitions that syntactically seem proper might in fact be degenerate by e.g. including arguments such as $(x : \mathbf{0}) \to B(!_A(x))$. Even worse, to detect such "false" arguments, we would need to check if a type is empty or not, which is well-known to be undecidable (by reduction from the Halting Problem, for instance).

Finitary inductive-inductive definitions, i.e. definitions where the premises of inductive arguments are isomorphic to $\mathsf{Fin}(m)$ for some $m : \mathbb{N}$, are graded. Note that all examples of proper inductive-inductive definitions we have considered so far indeed have been finitary.

Proposition 5.31 Every finitary inductive-inductive definition $(\gamma_{\mathsf{A}}, \gamma_{\mathsf{B}})$ gives rise to a graded inductive-inductive container (C_A, C_B) whose extension $[\![(C_A, C_B)]\!]$ is naturally isomorphic to $\mathsf{Arg}_{\gamma_{\mathsf{A}}, \gamma_{\mathsf{B}}}$, i.e.

$$[\![(C_A, C_B)]\!](X, Y, \mathsf{intro_A}) \cong \mathsf{Arg}_{\gamma_{\mathsf{A}}, \gamma_{\mathsf{B}}}(X, Y, \mathsf{intro_A}) \ ,$$

for all $(X, Y, \mathsf{intro}_{\mathrm{A}})$ in $\mathsf{Dialg}([\![C_A]\!], U)$ (naturally in $(X, Y, \mathsf{intro}_{\mathrm{A}})$).

Proof. We know by Corollaries 5.19 and 5.25 that $(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}})$ can be written in the form (5.1) and (5.2), e.g.

$$\gamma'_{\mathrm{A}} = \mathsf{non\text{-}ind}(S^A, \lambda s.\, \mathsf{A\text{-}ind}(P^A_{\mathrm{A}}(s), \mathsf{B\text{-}ind}(P^A_{\mathrm{B}}(s), \mathsf{inr} \circ h_{\mathrm{index}}(s), \mathsf{nil})))$$

and

$$\begin{aligned}\gamma'_{\mathrm{B}} = \mathsf{non\text{-}ind}(S^B, \lambda s.\, &\mathsf{A\text{-}ind}(P^B_{\mathrm{A}}(s), \\ &\underbrace{\mathsf{B\text{-}ind}(P^B_{\mathrm{B},0}(s), h_0(s), \ldots \mathsf{B\text{-}ind}(P^B_{\mathrm{B},n(s)}(s), h_{n(s)}(s), \mathsf{nil}(a(s))))))}_{n(s)\text{ many}}\ .\end{aligned}$$

We now define a rank function $r : \mathsf{A\text{-}Term}(\gamma'_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}) \to \mathbb{N}$ by

$$\begin{aligned} r(\mathsf{a}_{\mathrm{ref}}(x)) &= 0 \\ r(\mathsf{b}_{\mathrm{ref}}(x)) &= 0 \\ r(\mathsf{arg}(\langle s, \langle j_A, \langle j_B, \star\rangle\rangle\rangle)) &= \mathsf{suc}(\max_{i:P^A_{\mathrm{A}}(s)} r(j_A(i))) \end{aligned}$$

This maximum is only well-defined because we know $P^A_{\mathrm{A}}(s)$ is finite. Intuitively, $r(x)$ is the number of constructors used in the index encoded by x.

We now define

$$n^{gB}(s) := \max_{i:\mathsf{Fin}(n(s)), x:P^B_{\mathrm{B},i}(s)} r(h^B(s, i, x))$$

which once again is well-defined because $(\gamma'_{\mathrm{A}}, \gamma'_{\mathrm{B}})$ is finitary. As an abbreviation, let us write $P^B_{\mathrm{B}}(s) := P^B_{\mathrm{B},0}(s) + \ldots + P^B_{\mathrm{B},n(s)}(s)$. Let

$$f_0 : \{x : \mathsf{A\text{-}Term}(\gamma'_{\mathrm{A}}, P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s)) \mid r(x) = 0\} \to P^B_{\mathrm{A}}(s) + \mathbf{1}$$

be the function which maps $\mathsf{a}_{\mathrm{ref}}(x)$ to $\mathsf{inl}(x)$ and everything else to $\mathsf{inr}(\star)$. We define $P^{gB}_{\mathrm{B}}(s, 0) : P^B_{\mathrm{A}}(s) \to \mathsf{Set}$ by

$$P^{gB}_{\mathrm{B}}(s, 0, y) := \{x : P^B_{\mathrm{B}}(s) \mid r([h_0(s), \ldots, h_{n(s)}(s)](x)) = 0 \wedge f_0(x) = \mathsf{inl}(y)\}\ .$$

Similarly, we can define

$$f_i : \{x : \mathsf{A\text{-}Term}(\gamma'_{\mathrm{A}}, P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s)) \mid r(x) = i\} \to [\![\gamma'_{\mathrm{A}}]\!]^i(P^B_{\mathrm{A}}(s), P^{gB}_{\mathrm{A}}(s, 0), \ldots, P^{gB}_{\mathrm{A}}(s, i{-}1)) + \mathbf{1}$$

and

$$P^{gB}_{\mathrm{B}}(s, i, y) := \{x : P^B_{\mathrm{B}}(s) \mid r([h_0(s), \ldots, h_{n(s)}(s)](x)) = i \wedge f_i(x) = \mathsf{inl}(y)\}\ .$$

We finally define $a^{gB}(s) := f_{r(a(s))}(a(s))$. It should be clear that the decoding of γ'_{B}, which is isomorphic to the decoding of γ_{B}, is isomorphic to the decoding of $(S^B, P^B_{\mathrm{A}}, n^{gB}, P^{gB}_{\mathrm{B}}, a^{gB})$. □

5.3 Reduction to extensional indexed inductive definitions

How strong is the theory of inductive-inductive definitions? Indexed inductive definitions naturally embed into inductive-inductive definitions: The indexed inductive definition $X : I \to \mathsf{Set}$ can be regarded as an inductive-inductive definition of $I' : \mathsf{Set}$ and $X : I' \to \mathsf{Set}$ where I' is an isomorphic copy of I, i.e. given by one constructor $\mathsf{intro}_{I'} : I \to I'$ (see Section 3.2.4.2). Hence inductive-inductive definitions are at least as strong as the theory of indexed inductive definitions. We will now sharpen this result by showing that the theory of inductive-inductive definitions, with the simple elimination rules from Section 3.2.5.2, can be interpreted in the extensional theory of indexed inductive definitions. Thus, if there is any difference in proof-theoretical strength between the two theories, it is either because of extensionality (which is unlikely, as models of Type Theory which gives upper bounds for its proof theoretical strength usually also interpret the the equality reflection rule [Setzer, 1996]), or the general elimination rules presented in Chapter 4.

The general idea of the reduction is to first define "pre-sets" $\mathsf{pre}A : \mathsf{Set}$ and $\mathsf{pre}B : \mathsf{Set}$ without index information. This makes it possible to define $\mathsf{pre}A$ and $\mathsf{pre}B$ using an ordinary mutual definition (hence a 2-indexed definition), but the lack of precision means that we have also included junk into our sets. We thus define "goodness predicates" $\mathsf{good}A : \mathsf{pre}A \to \mathsf{Set}$ and $\mathsf{good}B : \mathsf{pre}A \to \mathsf{pre}B \to \mathsf{Set}$ which singles out the well-formed elements that respect the original index information. We can then define $A := (\Sigma x : \mathsf{pre}A)\mathsf{good}A(x)$ and $B(\langle x, g\rangle) := (\Sigma y : \mathsf{pre}B)\mathsf{good}B(x, y)$ and prove that the introduction and (simple) elimination rules are sound.

Before we prove the general theorem, let us consider an example. In fact, the reader is advised to understand this example (and the following Example 5.36) to get a general idea of the reduction before tackling the general case, which does not offer any additional technical difficulty.

Example 5.32 (Contexts and types as an indexed inductive definition) How would we represent the contexts and types from Example 3.1 if we did not have dependent types? A reasonable approach is to include possibly non-wellformed types, and then afterwards check that the types in question are well-formed. For this check to be possible, we still need to store the context, that we believe the type is well-formed in, in the type, and so, the definition of contexts and types is still simultaneous, although the typically inductive-inductive phenomenon of contexts appearing as indices of types has disappeared.

Thus, we take the original definition of contexts and types

$$\frac{}{\varepsilon : \mathsf{Ctxt}} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma)}{\Gamma \rhd \sigma : \mathsf{Ctxt}}$$

$$\frac{\Gamma : \mathsf{Ctxt}}{\iota_\Gamma : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma) \quad \tau : \mathsf{Ty}(\Gamma \rhd \sigma)}{\Pi_\Gamma(\sigma, \tau) : \mathsf{Ty}(\Gamma)}$$

and drop all index information, so that we end up with

$$\frac{}{\varepsilon_{\mathsf{pre}} : \mathsf{preCtxt}} \qquad \frac{\Gamma : \mathsf{preCtxt} \quad \sigma : \mathsf{preTy}}{\Gamma \rhd_{\mathsf{pre}} \sigma : \mathsf{preCtxt}}$$

$$\frac{\Gamma : \mathsf{preCtxt}}{\iota_{\mathsf{pre}}(\Gamma) : \mathsf{preTy}} \qquad \frac{\Gamma : \mathsf{preCtxt} \quad \sigma : \mathsf{preTy} \quad \tau : \mathsf{preTy}}{\Pi_{\mathsf{pre}}(\Gamma, \sigma, \tau) : \mathsf{preTy}}$$

Given a context $\Gamma : \mathsf{Ctxt}$ or type $\sigma : \mathsf{Ty}(\Delta)$, we can always erase all "type information" to get a corresponding precontext $\mathsf{pre}(\Gamma) : \mathsf{preCtxt}$ and pretype $\mathsf{pre}(\sigma) : \mathsf{preTy}$. It should be clear that there are plenty of precontexts and pretypes that are not the erasure of any proper contexts or types, though. Hence, we cannot expect preCtxt and preTy to satisfy the same induction principle as Ctxt and Ty, and interpreting the latter as the former would make the elimination principle unsound. We now define predicates $\mathsf{goodCtxt} : \mathsf{preCtxt} \to \mathsf{Set}$ and $\mathsf{goodTy} : \mathsf{preCtxt} \to \mathsf{preTy} \to \mathsf{Set}$ that are true exactly for those precontexts and pretypes that are actually well-formed; the proposition $\mathsf{goodCtxt}(\Gamma)$ is true if Γ is a well-formed context, i.e. it is the erasure of some $\Gamma' : \mathsf{Ctxt}$, and $\mathsf{goodTy}(\Gamma, \sigma)$ is true if σ is a well-formed type in context Γ, i.e. it is the erasure of some $\sigma' : \mathsf{Ty}(\Gamma')$ where Γ is the erasure of Γ'. We do this by reintroducing the index information, using a simultaneous indexed inductive definition:

$$\frac{}{\varepsilon_{\mathsf{good}} : \mathsf{goodCtxt}(\varepsilon_{\mathsf{pre}})}$$

$$\frac{\Gamma : \mathsf{preCtxt} \quad \widetilde{\Gamma} : \mathsf{goodCtxt}(\Gamma) \quad \sigma : \mathsf{preTy} \quad \widetilde{\sigma} : \mathsf{goodTy}(\Gamma, \sigma)}{(\Gamma, \widetilde{\Gamma}) \rhd_{\mathsf{good}} (\sigma, \widetilde{\sigma}) : \mathsf{goodCtxt}(\Gamma \rhd_{\mathsf{pre}} \sigma)}$$

$$\frac{\Gamma : \mathsf{preCtxt} \quad \widetilde{\Gamma} : \mathsf{goodCtxt}(\Gamma)}{\iota_{\mathsf{good}}(\Gamma, \widetilde{\Gamma}) : \mathsf{goodTy}(\Gamma, \iota_{\mathsf{pre}}(\Gamma))}$$

$$\frac{\begin{array}{ccc}\Gamma : \mathsf{preCtxt} & \sigma : \mathsf{preTy} & \tau : \mathsf{preTy} \\ \widetilde{\Gamma} : \mathsf{goodCtxt}(\Gamma) & \widetilde{\sigma} : \mathsf{goodTy}(\Gamma, \sigma) & \widetilde{\tau} : \mathsf{goodTy}(\Gamma \rhd_{\mathsf{pre}} \sigma)\end{array}}{\Pi_{\mathsf{good}}(\Gamma, \widetilde{\Gamma}, \sigma, \widetilde{\sigma}, \tau, \widetilde{\tau}) : \mathsf{goodTy}(\Gamma, \Pi_{\mathsf{pre}}(\Gamma, \sigma, \tau))}$$

For instance, $\Pi_{\mathsf{good}}(\Gamma, \widetilde{\Gamma}, \sigma, \widetilde{\sigma}, \tau, \widetilde{\tau})$ says that if Γ is a well-formed context, σ is a well-formed type in context Γ and τ is a well-formed type in context $\Gamma \rhd_{\mathsf{pre}} \sigma$, then $\Pi_{\mathsf{pre}}(\Gamma, \sigma, \tau)$ is a well-formed type in the original context Γ.

We now define the interpretation of Ctxt and Ty to be

$$\llbracket \mathsf{Ctxt} \rrbracket := (\Sigma \Gamma : \mathsf{preCtxt})\mathsf{goodCtxt}(\Gamma)$$
$$\llbracket \mathsf{Ty} \rrbracket(\langle \Gamma, \widetilde{\Gamma} \rangle) := (\Sigma \sigma : \mathsf{preTy})\mathsf{goodTy}(\Gamma, \sigma)$$

which shows that the formation rules are sound with respect to the translation. Furthermore, we can validate the introduction rules by pairing up the goodness proofs we have asked for:

$$\llbracket \varepsilon \rrbracket : \llbracket \mathsf{Ctxt} \rrbracket$$

$$[\![\varepsilon]\!] = \langle \varepsilon_{\mathsf{pre}}, \varepsilon_{\mathsf{good}} \rangle$$

$$\begin{aligned} [\![\triangleright]\!] &: (\Gamma : [\![\mathsf{Ctxt}]\!]) \to [\![\mathsf{Ty}]\!](\Gamma) \to [\![\mathsf{Ctxt}]\!] \\ \langle \Gamma, \widetilde{\Gamma} \rangle [\![\triangleright]\!] \langle \sigma, \widetilde{\sigma} \rangle &= \langle \Gamma \triangleright_{\mathsf{pre}} \sigma, (\Gamma, \widetilde{\Gamma}) \triangleright_{\mathsf{good}} (\sigma, \widetilde{\sigma}) \rangle \end{aligned}$$

$$\begin{aligned} [\![\iota]\!] &: (\Gamma : \mathsf{Ctxt}) \to [\![\mathsf{Ty}]\!](\Gamma) \\ [\![\iota]\!](\langle \Gamma, \widetilde{\Gamma} \rangle) &= \langle \iota_{\mathsf{pre}}(\Gamma), \iota_{\mathsf{good}}(\Gamma, \widetilde{\Gamma}) \rangle \end{aligned}$$

$$\begin{aligned} [\![\Pi]\!] &: (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (\sigma : [\![\mathsf{Ty}]\!](\Gamma)) \to [\![\mathsf{Ty}]\!](\Gamma [\![\triangleright]\!] \sigma) \to [\![\mathsf{Ty}]\!](\Gamma) \\ [\![\Pi]\!](\langle \Gamma, \widetilde{\Gamma} \rangle, \langle \sigma, \widetilde{\sigma} \rangle, \langle \tau, \widetilde{\tau} \rangle) &= \langle \Pi_{\mathsf{pre}}(\Gamma, \sigma, \tau), \Pi_{\mathsf{good}}(\Gamma, \widetilde{\Gamma}, \sigma, \widetilde{\sigma}, \tau, \widetilde{\tau}) \rangle \end{aligned}$$

We will get back to the elimination rules in Example 5.36. ■

We now consider a general inductive-inductive definition. In order to reduce complexity, we work with graded inductive-inductive containers from Section 5.2.3. Recall that extensionally, all finitary inductive-inductive definitions can be reduced to this form (and many non-finitary ones too). It should however be stressed that this normal form only is a convenience for the proof, and not a necessity – we simply choose to prove this particular version of the theorem in order to avoid induction over codes and syntactical clutter that obscures the idea behind the proof. Since the general version of the soundness theorem will require extensional equality anyway, we do not lose anything by immediately switching to the normal form.

Definition 5.33 Given a graded inductive-inductive container (C_A, C_B) where $C_A = (S^A, P^A_{\mathrm{A}}, P^A_{\mathrm{B}}, h^A_{\mathrm{index}})$ and $C_B = (S^B, P^B_{\mathrm{A}}, n^B, P^B_{\mathrm{B}}, a^B)$, the mutually inductive data types pre$A$: Set, pre$B$: Set are given by the following constructors:

$$\begin{aligned} \mathsf{in}_{\mathsf{pre}A} : {} & (s : S^A) \to \\ & (f : P^A_{\mathrm{A}}(s) \to \mathsf{pre}A) \to \\ & (g : P^A_{\mathrm{B}}(s) \to \mathsf{pre}B) \to \mathsf{pre}A \end{aligned}$$

$$\begin{aligned} \mathsf{in}_{\mathsf{pre}B} : {} & (s : S^B) \to \\ & (f : P^B_{\mathrm{A}}(s) \to \mathsf{pre}A) \to \\ & (g_0 : (x : P^B_{\mathrm{A}}(s)) \to P^B_{\mathrm{B}}(s, 0, x) \to \mathsf{pre}B) \to \\ & (g_1 : (x : [\![C_A]\!](P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s))) \to P^B_{\mathrm{B}}(s, 1, x) \to \mathsf{pre}B) \to \\ & \quad \vdots \\ & (g_{n^B(s)} : (x : [\![C_A]\!]^{n^B(s)}(P^B_{\mathrm{A}}(s), \vec{P^B_{\mathrm{B}}}(s))) \to P^B_{\mathrm{B}}(s, n^B(s), x) \to \mathsf{pre}B) \to \mathsf{pre}B \end{aligned}$$

Furthermore, the mutually indexed inductive data types good$A$: pre$A$ → Set and

$\mathsf{good}B : \mathsf{pre}A \to \mathsf{pre}B \to \mathsf{Set}$ are given by the following constructors:

$$
\begin{aligned}
\mathsf{in}_{\mathsf{good}A} : (s : S^A) \to\ & \\
& (f : P^A_{\mathrm{A}}(s) \to \mathsf{pre}A) \to \\
& (\tilde{f} : (x : P^A_{\mathrm{A}}(s)) \to \mathsf{good}A(f(x))) \to \\
& (g : P^A_{\mathrm{B}}(s) \to \mathsf{pre}B) \to \\
& (\tilde{g} : (P^A_{\mathrm{B}}(s)) \to \mathsf{good}B(f(h^A_{\mathrm{index}}(s,x)), g(x))) \to \mathsf{good}A(\mathsf{in}_{\mathsf{pre}A}(s,f,g))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{in}_{\mathsf{good}B} : (s : S^B) \to\ & \\
& (f : P^B_{\mathrm{A}}(s) \to \mathsf{pre}A) \to \\
& (\tilde{f} : (x : P^B_{\mathrm{A}}(s)) \to \mathsf{good}A(f(x))) \to \\
& (g_0 : (x : P^B_{\mathrm{A}}(s)) \to P^B_{\mathrm{B}}(s,0,x) \to \mathsf{pre}B) \to \\
& (\widetilde{g_0} : (x : P^B_{\mathrm{A}}(s)) \to (y : P^B_{\mathrm{B}}(s,0,x)) \to \mathsf{good}B(\overline{f}(s,0,x), g_0(x,y))) \to \\
& (g_1 : (x : [\![C_A]\!](P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s))) \to P^B_{\mathrm{B}}(s,1,x) \to \mathsf{pre}B) \to \\
& (\widetilde{g_1} : (x : [\![C_A]\!](P^B_{\mathrm{A}}(s), P^B_{\mathrm{B}}(s))) \to (y : P^B_{\mathrm{B}}(s,1,x)) \to \mathsf{good}B(\overline{f}(s,1,x), g_1(x,y))) \to \\
& \quad \vdots \\
& (g_{n^B(s)} : (x : [\![C_A]\!]^{n^B(s)}(P^B_{\mathrm{A}}(s), P^{\vec{B}}_{\mathrm{B}}(s))) \to P^B_{\mathrm{B}}(s, n^B(s), x) \to \mathsf{pre}B) \to \\
& (\widetilde{g_{n^B(s)}} : (x : [\![C_A]\!]^{n^B(s)}(P^B_{\mathrm{A}}(s), P^{\vec{B}}_{\mathrm{B}}(s))) \to (y : P^B_{\mathrm{B}}(s, n^B(s), x)) \to \\
& \qquad \mathsf{good}B(\overline{f}(s, n^B(s), x), g_{n^B(s)}(x,y))) \to \\
& \quad \mathsf{good}B([\overline{f}(s,0), \ldots, \overline{f}(s,n(s))](a(s)), \mathsf{in}_{\mathsf{pre}B}(s, f, g_0, \ldots, g_{n^B(s)})
\end{aligned}
$$

where

$$
\begin{aligned}
\overline{f}(s,0) &= f \\
\overline{f}(s,i+1) &= \mathsf{in}_{\mathsf{pre}A} \circ [\![C_A]\!]([\overline{f}(s,0), \ldots, \overline{f}(s,i)], [g_0, \ldots, g_i]) \ .
\end{aligned}
$$

■

The idea is the same as in Example 5.32: The sets $\mathsf{pre}A$ and $\mathsf{pre}B$ drops all index information, and $\mathsf{good}A$ and $\mathsf{good}B$ restore it (it might be helpful to compare Definition 5.33 and Definition 5.29).

We have presented the indexed inductive definitions as mutual inductively defined data types, but it should be clear that they can be presented straightforwardly as indexed containers or as codes in a system of indexed inductive definitions, with index set **2** (for a choice between $\mathsf{pre}A$ and $\mathsf{pre}B$). Before we can prove the soundness of the formation and introduction rules, we need a technical lemma:

Lemma 5.34 Let s, f, g_i and $\overline{f}$ be as in the arguments to $\mathsf{in}_{\mathsf{good}B}$ in Definition 5.33. Then

$$\mathsf{fst} \circ [\overline{f}(s,0), \ldots, \overline{f}(s,n(s))] = [\overline{\mathsf{fst} \circ f}(s,0), \ldots, \overline{\mathsf{fst} \circ f}(s,n(s))] \ .$$

□

Lemma 5.35 (Soundness of the formation and introduction rules) Let (C_A, C_B) be a graded container. If we define

$$\begin{aligned}\llbracket A \rrbracket &:= (\Sigma x : \mathsf{pre}A)\mathsf{good}A(x)\\ \llbracket B \rrbracket(\langle x, x_g \rangle) &:= (\Sigma y : \mathsf{pre}B)\mathsf{good}B(x, y) \ ,\end{aligned}$$

then constants $\mathsf{intro}_{\mathrm{A}}$ and $\mathsf{intro}_{\mathrm{B}}$ can be defined so that the introduction rules

$$\frac{x : \llbracket C_A \rrbracket(\llbracket A \rrbracket, \llbracket B \rrbracket)}{\mathsf{intro}_{\mathrm{A}}(x) : \llbracket A \rrbracket} \qquad \frac{x : \llbracket C_B \rrbracket(\llbracket A \rrbracket, \llbracket B \rrbracket, \mathsf{intro}_{\mathrm{A}})}{\mathsf{intro}_{\mathrm{B}}(x) : \llbracket B \rrbracket(\llbracket C_B \rrbracket_{\mathsf{Index}_{\mathrm{B}}}(x))}$$

are valid.

Proof. We begin by defining $\mathsf{intro}_{\mathrm{A}} : \llbracket C_A \rrbracket(\llbracket A \rrbracket, \llbracket B \rrbracket) \to \llbracket A \rrbracket$. By the definition of $\llbracket C_A \rrbracket$ and the η rules for Σ types, any $x : \llbracket C_A \rrbracket(\llbracket A \rrbracket, \llbracket B \rrbracket)$ is of the form $x = \langle s, \langle f, g \rangle \rangle$ where

$$\begin{aligned}&s : S^A\\ &f : P^A_{\mathrm{A}}(s) \to (\Sigma x : \mathsf{pre}A)\mathsf{good}A(x)\\ &g : (x : P^A_{\mathrm{B}}(s)) \to (\Sigma y : \mathsf{pre}B)\mathsf{good}B(\mathsf{fst}(f(h(x))), y) \ ,\end{aligned}$$

where we have expanded the definition of $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$. Thus we can define

$$\mathsf{intro}_{\mathrm{A}}(\langle s, \langle f, g \rangle \rangle) := \langle \mathsf{in}_{\mathsf{pre}A}(s, \mathsf{fst} \circ f, \mathsf{fst} \circ g), \mathsf{in}_{\mathsf{good}A}(s, \mathsf{fst} \circ f, \mathsf{snd} \circ f, \mathsf{fst} \circ g, \mathsf{snd} \circ g) \rangle$$

We can now do almost the same thing for $\mathsf{intro}_{\mathrm{B}}$: given $x = \langle s, \langle f, \langle g_0, \langle \ldots, g_{n^B(s)} \rangle \rangle \rangle \rangle$, we can use

$$\mathsf{in}_{\mathsf{pre}B}(s, \mathsf{fst} \circ f, \lambda x.\, \mathsf{fst} \circ g_0(x), \ldots, \lambda x.\, \mathsf{fst} \circ g_{n^B(s)}(x))$$

as the first component of $\mathsf{intro}_{\mathrm{B}}(\langle s, \langle f, \langle g_0, \langle \ldots, g_{n^B(s)} \rangle \rangle \rangle \rangle)$, and we would like to use

$$\mathsf{in}_{\mathsf{good}B}(s, \mathsf{fst} \circ f, \lambda x.\, \mathsf{fst} \circ g_0(x), \lambda x.\, \mathsf{snd} \circ g_0(x), \ldots, \lambda x.\, \mathsf{fst} \circ g_{n^B(s)}(x), \lambda x.\, \mathsf{snd} \circ g_{n^B(s)}(x))$$

as the second component, but the type is not obviously right: $\mathsf{ingood}B(\ldots)$ has type

$$\mathsf{good}B([\overline{\mathsf{fst} \circ f}(s, 0), \ldots, \overline{\mathsf{fst} \circ f}(s, n(s))](a(s)), \mathsf{in}_{\mathsf{pre}B}(\ldots)) \ ,$$

but needs to have type

$$\begin{aligned}&\mathsf{good}B(\mathsf{fst}(\llbracket C_B \rrbracket_{\mathsf{Index}_{\mathrm{B}}}(x)), \mathsf{in}_{\mathsf{pre}B}(\ldots)) =\\ &\qquad\qquad\qquad \mathsf{good}B(\mathsf{fst}([\overline{f}(s, 0), \ldots, \overline{f}(s, n(s))](a(s))), \mathsf{in}_{\mathsf{pre}B}(\ldots))\end{aligned}$$

By Lemma 5.34 and the equality reflection principle, these two types are equal and we can define

$$\begin{aligned}&\mathsf{intro}_{\mathrm{B}}(\langle s, \langle f, \langle g_0, \ldots, g_{n^B(s)} \rangle \rangle \rangle) :=\\ &\quad \langle \mathsf{in}_{\mathsf{pre}B}(s, \mathsf{fst} \circ f, \lambda x.\, \mathsf{fst} \circ g_0(x), \ldots, \lambda x.\, \mathsf{fst} \circ g_{n^B(s)}(x)),\\ &\quad\ \mathsf{in}_{\mathsf{good}B}(s, \mathsf{fst} \circ f, \lambda x.\, \mathsf{fst} \circ g_0(x), \lambda x.\, \mathsf{snd} \circ g_0(x),\\ &\qquad\qquad\qquad \ldots,\\ &\qquad\qquad\qquad \lambda x.\, \mathsf{fst} \circ g_{n^B(s)}(x), \lambda x.\, \mathsf{snd} \circ g_{n^B(s)}(x)) \rangle \ . \quad \square\end{aligned}$$

We would also like to show that the elimination rules are valid for $[\![A]\!]$ and $[\![B]\!]$. We do this for the simple elimination rules from Section 3.2.5.2. Let us once again start with a concrete example.

Example 5.36 (Elimination rules for contexts and types) For the contexts and types from Example 5.32, we would like to define elimination constants $\mathsf{elim}_{[\![\mathsf{Ctxt}]\!]}$ and $\mathsf{elim}_{[\![\mathsf{Ty}]\!]}$ of type

$$
\begin{aligned}
\mathsf{elim}_{\mathsf{Ctxt}} : {} & (P : [\![\mathsf{Ctxt}]\!] \to \mathsf{Set}) \to (Q : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to [\![\mathsf{Ty}]\!](\Gamma) \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\varepsilon} : P([\![\varepsilon]\!])) \to \\
& (\mathsf{step}_{\triangleright} : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (\sigma : [\![\mathsf{Ty}]\!](\Gamma)) \to P(\Gamma) \to Q(\Gamma, \sigma) \to P(\Gamma[\![\triangleright]\!]\sigma)) \to \\
& (\mathsf{step}_{\Pi} : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (\sigma : [\![\mathsf{Ty}]\!](\Gamma)) \to (\tau : [\![\mathsf{Ty}]\!](\Gamma[\![\triangleright]\!]\sigma)) \to P(\Gamma) \\
& \qquad \to Q(\Gamma, \sigma) \to Q(\Gamma[\![\triangleright]\!]\sigma, \tau) \to Q(\Gamma, [\![\Pi]\!](\Gamma, \sigma, \tau))) \to \\
& (\mathsf{step}_{\iota} : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to P(\Gamma) \to Q(\Gamma, [\![\iota]\!](\Gamma))) \to \\
& (\Gamma : [\![\mathsf{Ctxt}]\!]) \to P(\Gamma)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{elim}_{\mathsf{Ty}} : {} & (P : [\![\mathsf{Ctxt}]\!] \to \mathsf{Set}) \to (Q : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to [\![\mathsf{Ty}]\!](\Gamma) \to \mathsf{Set}) \to \\
& (\mathsf{step}_{\varepsilon} : P([\![\varepsilon]\!])) \to \\
& (\mathsf{step}_{\triangleright} : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (\sigma : [\![\mathsf{Ty}]\!](\Gamma)) \to P(\Gamma) \to Q(\Gamma, \sigma) \to P(\Gamma[\![\triangleright]\!]\sigma)) \to \\
& (\mathsf{step}_{\Pi} : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (\sigma : [\![\mathsf{Ty}]\!](\Gamma)) \to (\tau : [\![\mathsf{Ty}]\!](\Gamma[\![\triangleright]\!]\sigma)) \to P(\Gamma) \\
& \qquad \to Q(\Gamma, \sigma) \to Q(\Gamma[\![\triangleright]\!]\sigma, \tau) \to Q(\Gamma, [\![\Pi]\!](\Gamma, \sigma, \tau))) \to \\
& (\mathsf{step}_{\iota} : (\Gamma : [\![\mathsf{Ctxt}]\!]) \to P(\Gamma) \to Q(\Gamma, [\![\iota]\!](\Gamma))) \to \\
& (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (\sigma : [\![\mathsf{Ty}]\!](\Gamma)) \to Q(\Gamma, \sigma)
\end{aligned}
$$

The high-level idea is to use the elimination principle for goodCtxt and goodTy for this – indeed, the way we have defined the interpretation of the constructors $[\![\iota]\!]$, $[\![\Pi]\!]$ etc., the step functions for $\mathsf{elim}_{\mathsf{Ctxt}}$ and $\mathsf{elim}_{\mathsf{Ty}}$ above are basically uncurried versions of the step functions for $\mathsf{elim}_{\mathsf{goodCtxt}}$ and $\mathsf{elim}_{\mathsf{goodTy}}$. We will face and overcome two problems.

The first problem we meet almost directly. We would like to implement $\mathsf{elim}_{\mathsf{Ctxt}}$ and $\mathsf{elim}_{\mathsf{Ty}}$ in terms of $\mathsf{elim}_{\mathsf{goodCtxt}}$ and $\mathsf{elim}_{\mathsf{goodTy}}$ respectively. Let us focus on $\mathsf{elim}_{\mathsf{Ctxt}}$. Let all the arguments to $\mathsf{elim}_{\mathsf{Ctxt}}$ be given, in particular P and Q of types

$$
\begin{aligned}
& P : \big((\Sigma\Gamma : \mathsf{preCtxt})\mathsf{goodCtxt}(\Gamma)\big) \to \mathsf{Set} \\
& Q : (\langle \Gamma, \Gamma_g \rangle : (\Sigma\Gamma : \mathsf{preCtxt})\mathsf{goodCtxt}(\Gamma)) \to \big((\Sigma\sigma : \mathsf{preTy})\mathsf{goodTy}(\Gamma, \sigma)\big) \to \mathsf{Set}
\end{aligned}
$$

The motives for $\mathsf{elim}_{\mathsf{goodCtxt}}$, on the other hand, are of the form

$$
\begin{aligned}
& P' : (\Gamma : \mathsf{preCtxt}) \to \mathsf{goodCtxt}(\Gamma) \to \mathsf{Set} \\
& Q' : (\Gamma : \mathsf{preCtxt}) \to (\sigma : \mathsf{preTy}) \to \mathsf{goodTy}(\Gamma, \sigma) \to \mathsf{Set}
\end{aligned}
$$

We can choose $P'(\Gamma, \Gamma_g) = P(\langle \Gamma, \Gamma_G \rangle)$, but for Q', we have what appears to be a problem: we have no $\Gamma_g : \mathsf{goodCtxt}(\Gamma)$ to give to Q! Luckily, we can extract such a goodness proof from σ_g : goodTy(Γ, σ), which we do have. Inspecting the constructors $\iota_{\text{good}}(\Gamma, \widetilde{\Gamma})$ and $\Pi_{\text{good}}(\Gamma, \widetilde{\Gamma}, \sigma, \widetilde{\sigma}, \tau, \widetilde{\tau})$, they both contain a goodness proof $\widetilde{\Gamma}$: goodCtxt(Γ) for the current context. Hence, by the induction principle for goodTy, we get such a proof extractGood$_\Gamma(\sigma_g)$: goodCtxt(Γ) for all σ_g : goodTy(Γ, σ). Thus, we can choose

$$P'(\Gamma, \Gamma_g) \coloneqq P(\langle \Gamma, \Gamma_G \rangle)$$
$$Q'(\Gamma, \sigma, \sigma_g) \coloneqq Q(\langle \Gamma, \text{extractGood}_\Gamma(\sigma_g) \rangle, \langle \sigma, \sigma_g \rangle)$$

The step functions we have are now curried versions of the step functions we need, but there is a second problem: with the motive we have chosen, for Γ, Γ_g, σ and σ_g of appropriate type we get recursive calls of type $Q(\langle \Gamma, \text{extractGood}_\Gamma(\sigma_g) \rangle, \langle \sigma, \sigma_g \rangle)$, but the elimination principle we want to implement expects recursive calls of type $Q(\langle \Gamma, \Gamma_g \rangle, \langle \sigma, \sigma_g \rangle)$, i.e. using the given goodness proof Γ_g instead of the reconstructed goodness proof extractGood$_\Gamma(\sigma_g)$. Also this problem can be overcome, this time by noticing that in fact, all goodness proof of a given type are equal, hence also extractGood$_\Gamma(\sigma_g)$ and Γ_g. Also this can be proven using the elimination principle for goodTy and goodCtxt; this time, a simultaneous induction is necessary. By the equality reflection principle, the uncurried step functions have the right type and we have succeeded in defining elim$_{\text{Ctxt}}$ and elim$_{\text{Ty}}$.

The computation rules for elim$_{\text{Ctxt}}$ and elim$_{\text{Ty}}$ follow from the computation rules for elim$_{\text{goodCtxt}}$ and elim$_{\text{goodTy}}$ and the equality reflection rule again. ■

Armed with the experiences from the example, we can prove in general:

Lemma 5.37 (Extracted, unique goodness)

(i) If y_g : good$B(x, y)$ then there is a term extractGood$_x(y_g)$: good$A(x)$.

(ii) If x_g, x'_g : good$A(x)$ then there is $p : x_g \equiv_{\text{good}A(x)} x'_g$.

Proof. Both statements follow by an easy application of the elimination rules for good$A$ and good$B$. For (i), we use the motive $P(x, x_g) = \mathbf{1}$, $Q(x, y, y_g) = \text{good}A(x)$, i.e. we only do induction on good$B$. For (ii), we use the motive $P(x, x_g) = (x'_g : \text{good}A(x)) \to x_g \equiv x'_g$ and $Q(x, y, y_g) = (y'_g : \text{good}B(x, y)) \to y_g \equiv y'_g$. □

Using this, we can prove the soundness of the elimination rules for a general graded inductive-inductive container exactly as in Example 5.36.

Lemma 5.38 (Soundness of the simple elimination rules) Let (C_A, C_B) be a graded container, and define $[\![A]\!]$, $[\![B]\!]$, intro$_{\text{A}}$ and intro$_{\text{B}}$ as in Lemma 5.35. Also constants elim$_A$ and elim$_B$ can be defined which validates the elimination and computation rules. □

Since the data types pre$A$, pre$B$, good$A$ and good$B$ are constructed using indexed inductive definitions only, we have, in proof-theoretical terms, (almost) constructed a reduction from the theory of inductive-inductive definitions to the theory of indexed inductive definitions. It is not quite a reduction, since we have not interpreted the rules dealing with the large types SP$_{\text{A}}$ and SP$_{\text{B}}$. It is very possible that these could be coded in the large type IID somehow. However, from a practical or implementation point of view, this is irrelevant, as we have successfully dealt with the hard part of the theory, namely the introduction and elimination rules for A_{γ_A,γ_B} and B_{γ_A,γ_B} – there is no harm in having a large type of codes around if decoding them takes no extra effort. We summarise the development of this section in a theorem:

Theorem 5.39 The (extensional) theory of inductive-inductive definitions with simple elimination rules can be interpreted in the extensional theory of indexed inductive definitions combined with the formation and introduction rules for SP_A and SP_B. □

5.4 Summary and discussion

In this chapter, we have justified the existence of inductive-inductive definitions in two different ways: first by constructing a model in classical set theory, and then by a translation to a more well-known type theory.

The set-theoretical model is quite standard. The inductive-inductive definitions are modelled as inductive definitions are usually modelled in set theory, i.e. by iterating a monotone operator until a fixed point is reached. Both A and $B : A \to \mathsf{Set}$ are generated at the same time, and since set theory is untyped, it does not matter that A appears in the "type" of B.

The second justification is more satisfying from a constructive point of view. It can also, if one so wishes, be seen as a model construction, or alternatively as a proof-theoretical reduction.

Constructive models The set-theoretical model we have constructed lives in ZFC set theory + the existence of two inaccessible cardinals. This is mostly for convenience, as it allows us to reuse results from set theory without worrying if they apply in our setting or not. It should be clear that a considerably weaker theory is enough to interpret inductive-inductive definitions set-theoretically, with a reasonable guess being CZF + REA [Aczel and Rathjen, 2010]. Since we are going to extend the model presented here to cover also inductive-recursive definitions in Section 6.1, and this will require considerable more strength, we do not feel so bad about the currently far too strong theory used.

Translating codes not in container form The reduction to indexed inductive definitions was only given for codes in "container normal form". This is not a technical restriction, but rather a pedagogical one; a general treatment would necessarily do induction over the codes (as indeed the reduction to the normal form does), which would make the interpretation of especially the elimination rules unnecessarily hard to follow. As Examples 5.32 and 5.36 shows, it is perfectly possible to translate definitions not in normal form using the same recipe.

The need for extensionality The translation of inductive-inductive definitions to indexed inductive definitions takes place in extensional Type Theory, and we have made full use of this by applying the equality reflection rule in the interpretation of both the introduction and the elimination rules. It is not hard to replace these uses with explicit coercions using subst instead. This way, the introduction and elimination rules can be interpreted in *intensional* Type Theory with indexed inductive definitions. However, the computation rules are still only valid up to propositional equality, even for closed codes. One possible solution to this problem, suggested by Conor McBride (private communication) is to use a propositional universe as discussed in 2.1.7 for the goodness proofs.

Interpreting the general elimination rules The translation from inductive-inductive definitions to indexed inductive definitions only worked for the simple elimination rules from 3.2.5.2, and not the general elimination rules. The reason is simple: the elimination rules of the target theory does not support the "recursive-recursive" nature of the inductive-inductive elimination rules, where the second component of the motive

$$Q : (x : A) \to B(x) \to P(x) \to \mathsf{Set}$$

depends on the first component $P : A \to \mathsf{Set}$. If one were to add these kind of elimination rules to indexed inductive definitions, then the correspondence between the two theories would be exact.

Chapter 6

Extensions

Contents

In this chapter, we consider two orthogonal extensions of the theory of inductive-inductive definitions. Both are natural from a user perspective, and have indeed been used together by e.g. Danielsson [2007]. In Section 6.1, we combine the theory of inductive-inductive definitions and the theory of inductive-recursive definitions into the theory of inductive-inductive-recursive definitions. We extend Dybjer and Setzer's [1999] model construction and combine it with the model construction in Section 5.1 to show that the combined theory is sound. Finally, in Section 6.2 we explore how we can allow an inductive-inductive definition of telescopes

$$\begin{aligned} &A : \mathsf{Set}, \\ &B : A \to \mathsf{Set}, \\ &C : (x : A) \to B(x) \to \mathsf{Set} \\ &\quad\vdots \end{aligned}$$

of more than two levels, as well as more general "families" such as

$$\begin{aligned} &A : \mathsf{Set}, \\ &B : (A \times A) \to \mathsf{Set} \end{aligned}$$

or

$$B : (\mathbb{N} \times A) \to \mathsf{Set}$$

These extensions are justified via the categorical semantics in Chapter 4. This chapter gets us closer to a formalisation of all the kinds of definitions used by e.g. Danielsson [2007], but not all the way there. For instance, yet another extension would be needed to allow later constructors to depend on earlier ones for the same data type.

6.1 Inductive-inductive-recursive definitions

In inductive-inductive definitions, a set A is defined inductively simultaneously with an inductive family $B : A \to \mathsf{Set}$. In inductive-recursive definitions, $B : A \to \mathsf{Set}$ is instead defined recursively. But what if we need both an inductively defined $B_1 : A \to \mathsf{Set}$ and a recursively defined $B_2 : A \to \mathsf{Set}$ the same time? We now present an axiomatisation of inductive-inductive-recursive definitions, which allow the simultaneous definition of

$$\begin{array}{ll} A : \mathsf{Set} & \text{(inductively)} \\ B : A \to \mathsf{Set} & \text{(inductively)} \\ T : A \to D & \text{(recursively)} \end{array}$$

Such definitions were used by Danielsson [2007] to formalise the well-typed syntax of Type Theory (defined inductively), together with a hereditary substitution operation (defined recursively).

Example 6.1 (Danielsson [2007]) We informally present a simplified account of the first few levels of Danielsson's construction, extending Example 3.1. The contexts and the types are as in Example 3.1, i.e. we have a an empty context ε, a context extension operation $\rhd$, a base type ι_Γ in each context Γ and dependent function types $\Pi_\Gamma(\sigma, \tau)$. On top of this, we add inductively defined substitutions $\mathsf{Sub} : \mathsf{Ctxt} \to \mathsf{Ctxt} \to \mathsf{Set}$; their exact form is not important for our purposes, except that we will need them to include a "lifting" operation

$$\uparrow_{\Gamma,\Delta} : (\rho : \mathsf{Sub}(\Gamma, \Delta)) \to (\sigma : \mathsf{Ty}(\Gamma)) \to \mathsf{Sub}(\Gamma \rhd \sigma, \Delta \rhd (\sigma/\rho))$$

often written infix, which lifts a substitution ρ to an extended context with a new variable by acting like ρ on the old variables and mapping the new variable to itself. In the type of $\uparrow$, the function $/$ is the application of a substitution that we now will define. Notice that this makes the definition very simultaneous indeed.

The function $/_{\Gamma,\Delta} : \mathsf{Ty}(\Gamma) \to \mathsf{Sub}(\Gamma, \Delta) \to \mathsf{Ty}(\Delta)$ is defined by recursion over $\mathsf{Ty}(\Gamma)$, written infix and with Γ, Δ implicit:

$$\begin{aligned} \iota_\Gamma/\rho &= \iota_\Delta \\ \Pi_\Gamma(\sigma, \tau)/\rho &= \Pi_\Delta(\sigma/\rho, \tau/(\rho \uparrow \sigma)) \end{aligned}$$

We will not be able to support this example fully with our axiomatisation, partly because we need more than two levels (such an extension will be given in Section 6.2), but mostly because the codomain of $/_{\Gamma,\Delta}$ is $\mathsf{Ty}(\Delta)$, which is defined at the same time as $/_{\Gamma,\Delta}$, whereas we require the codomain D of recursive functions to be a previously introduced type. Since constructors are mapped to constructors, this does not seem to offer any foundational difficulties. This development should be seen as a first step towards a theory that can justify Danielsson's construction completely. ■

We will use IIR as an abbreviation for inductive-inductive-recursive definitions. Unfortunately, this abbreviation is also used for indexed inductive-recursive definitions, but we hope that no confusion will arise, as no such definitions occur in this thesis.

6.1.1 The axiomatisation of inductive-inductive-recursive definitions

The idea behind the axiomatisation is to combine the universe of codes for inductive-inductive definitions presented in Section 3.2.3 with Dybjer and Setzer's universe of codes for inductive-recursive definitions presented in Section 3.2.2. We will follow the main design of the system for inductive-inductive definitions, while at the same time incorporate parts of the system for inductive-recursive definitions. The reader is invited to keep the development in Section 3.2.3 in mind, as there will be many similarities. The whole construction is parameterised by a (possible large) type D, the codomain of the recursively defined $T : A \to D$. We will suppress the premise D type from the rules that follow.

Looking back at the axiomatisation of inductive-inductive definitions, we see that we make use the functorial action of $\mathsf{Arg}^0_\mathsf{A}$ on morphisms. This will cause us some trouble, as we mentioned in Section 3.2.2 that the functorial action of Arg_IR requires extensional equality. Luckily, we can get away with function extensionality only and will thus require that we have it for the rest of this section. In Section 6.1.2, we will define subsystems of our system corresponding to "normal" inductive-inductive and inductive-recursive definitions, and both these subsystems will not require even function extensionality.

6.1.1.1 The universe $\mathsf{SP}^0_{\mathsf{IIR,A}}$ of descriptions of A

The universe of codes $\mathsf{SP}_{\mathsf{IIR,A}}$ is quite similar to the universe SP_A. The formation rule is the same:

$$\frac{X_\mathrm{ref} : \mathsf{Set}}{\mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})\ \mathsf{type}}$$

and we have the same codes, with only nil and A-ind different:

$$\frac{d : D}{\mathsf{nil}(d) : \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})} \qquad \frac{K : \mathsf{Set} \qquad \gamma : K \to \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})}{\mathsf{non\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})}$$

$$\frac{K : \mathsf{Set} \qquad \gamma : (K \to D) \to \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref} + K)}{\mathsf{A\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})}$$

$$\frac{K : \mathsf{Set} \qquad h_\mathrm{index} : K \to X_\mathrm{ref} \qquad \gamma : \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})}{\mathsf{B\text{-}ind}(K, h_\mathrm{index}, \gamma) : \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref})}$$

Compared to A-ind from $\mathsf{SP}_\mathsf{A}(X_\mathrm{ref})$, the difference is that the rest of the arguments, represented by $\gamma : (K \to D) \to \mathsf{SP}_{\mathsf{IIR,A}}(X_\mathrm{ref} + K)$, now can depend on T applied to the inductive argument as well. Just like for inductive-recursive definitions, the base case nil also needs to contain an element $d : D$ to be used as the value of T applied to the constructor.

For the decoding, also $\mathsf{Arg}_{\mathsf{IIR,A}}$ has almost the same formation rule as $\mathsf{Arg}_{\mathsf{A}}$, except we now also require a $Q : X \to D$ to use for the recursive family:

$$\frac{X_{\mathrm{ref}} : \mathsf{Set} \qquad \gamma : \mathsf{SP}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}) \qquad \begin{array}{c} X : \mathsf{Set} \\ Y : X \to \mathsf{Set} \\ Q : X \to D \end{array} \qquad \mathsf{rep}_{\mathsf{X}} : X_{\mathrm{ref}} \to X}{\mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathsf{X}}) : \mathsf{Set}}$$

Also the equations for $\mathsf{Arg}_{\mathsf{IIR,A}}$ are similar to the equations for $\mathsf{Arg}_{\mathsf{A}}$, with the only interesting difference being in the A-ind case: we define

$$\begin{aligned} &\mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K,\gamma), X, Y, Q, \mathsf{rep}_{\mathsf{X}}) = \\ &\qquad (\Sigma j : K \to X)\mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}} + K, \gamma(Q \circ j), X, Y, Q, [\mathsf{rep}_{\mathsf{X}}, j]) \end{aligned}$$

which can be compared with

$$\mathsf{Arg}_{\mathsf{A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K,\gamma), X, Y, \mathsf{rep}_{\mathsf{X}}) = (\Sigma j : K \to X)\mathsf{Arg}_{\mathsf{A}}(X_{\mathrm{ref}} + K, \gamma, X, Y, [\mathsf{rep}_{\mathsf{X}}, j])$$

and

$$\mathsf{Arg}_{\mathsf{IR}}(\delta(K,\gamma), X, Q) = (\Sigma g : K \to X)\mathsf{Arg}_{\mathsf{IR}}(\gamma(Q \circ g), X, Q) \ .$$

The full definition of $\mathsf{Arg}_{\mathsf{IIR,A}}$ is as follows, where we have once again written arguments that only get passed on in the recursive call as "_":

$$\begin{aligned} \mathsf{Arg}_{\mathsf{IIR,A}}(\_, \mathsf{nil}(d), \_, \_, \_, \_) &= \mathbf{1} \\ \mathsf{Arg}_{\mathsf{IIR,A}}(\_, \mathsf{non\text{-}ind}(K,\gamma), \_, \_, \_, \_) &= (\Sigma x : K)\mathsf{Arg}_{\mathsf{IIR,A}}(\_, \gamma(x), \_, \_, \_, \_) \\ \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K,\gamma), X, \_, Q, \mathsf{rep}_{\mathsf{X}}) &= \\ &(\Sigma j : K \to X)\mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}} + K, \gamma(Q \circ j), \_, \_, \_, [\mathsf{rep}_{\mathsf{X}}, j]) \\ \mathsf{Arg}_{\mathsf{IIR,A}}(\_, \mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma), \_, Y, \_, \mathsf{rep}_{\mathsf{X}}) &= \\ &((x : K) \to Y((\mathsf{rep}_{\mathsf{X}} \circ h_{\mathrm{index}})(x))) \times \mathsf{Arg}_{\mathsf{IIR,A}}(\_, \gamma, \_, \_, \_, \_) \end{aligned}$$

Like before, we define

$$\mathsf{Arg}^{0}_{\mathsf{IIR,A}} : \mathsf{SP}^{0}_{\mathsf{IIR,A}} \to (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to (Q : X \to \mathsf{Set}) \to \mathsf{Set}$$

by $\mathsf{Arg}^{0}_{\mathsf{IIR,A}}(\gamma, X, Y, Q) := \mathsf{Arg}_{\mathsf{IIR,A}}(\mathbf{0}, \gamma, X, Y, Q, !_X)$.

We also need to define the "recursive part" $\mathsf{Fun}_{\mathsf{IIR}}$, which corresponds to $\mathsf{Fun}_{\mathsf{IR}}$ for inductive-recursive definitions:

$$\frac{\begin{array}{c} X_{\mathrm{ref}} : \mathsf{Set} \\ \gamma : \mathsf{SP}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}) \end{array} \quad \begin{array}{c} X : \mathsf{Set} \\ Y : X \to \mathsf{Set} \\ Q : X \to D \end{array} \quad \mathsf{rep}_{\mathsf{X}} : X_{\mathrm{ref}} \to X \quad x : \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}, \gamma, X, Y, Q)}{\mathsf{Fun}_{\mathsf{IIR}}(X_{\mathrm{ref}}, \gamma, X, Y, Q, \mathsf{rep}_{\mathsf{X}}, x) : D}$$

We have the following defining equations, where we have written "_" for arguments handled in the same way as in the equations for $\mathsf{Arg}_{\mathsf{IIR,A}}$:

$$\begin{aligned}
\mathsf{Fun}_{\mathsf{IIR}}(\_, \mathsf{nil}(d), \_, \_, \_, \_, \star) &= d \\
\mathsf{Fun}_{\mathsf{IIR}}(\_, \mathsf{non\text{-}ind}(K,\gamma), \_, \_, \_, \_, \langle k, x\rangle) &= \mathsf{Fun}_{\mathsf{IR}}(\_, \gamma(k), \_, \_, \_, \_x) \\
\mathsf{Fun}_{\mathsf{IIR}}(\_, \mathsf{A\text{-}ind}(K,\gamma), \_, \_, Q, \_, \langle j, x\rangle) &= \mathsf{Fun}_{\mathsf{IIR}}(\_, \gamma(Q \circ j), \_, \_, \_, \_, x) \\
\mathsf{Fun}_{\mathsf{IIR}}(\_, \mathsf{B\text{-}ind}(K, h_{\text{index}}, \gamma), \_, \_, \_, \_, \langle j, x\rangle) &= \mathsf{Fun}_{\mathsf{IIR}}(\_, \gamma, \_, \_, \_, \_, x)
\end{aligned}$$

We define

$$\mathsf{Fun}^0_{\mathsf{IIR}} : (\gamma : \mathsf{SP}^0_{\mathsf{IIR,A}}) \to (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to (Q : X \to D) \to \\ \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X, Y, Q) \to D$$

by $\mathsf{Fun}^0_{\mathsf{IIR}}(\gamma, X, Y, Q, x) \coloneqq \mathsf{Fun}_{\mathsf{IIR}}(\mathbf{0}, \gamma, X, Y, Q, !_X, x)$.

6.1.1.2 Towards descriptions of B

Following the axiomatisation in Section 3.2.3.2, we would next like to define an action of $\mathsf{Arg}_{\mathsf{IIR,A}}$ on morphisms. What category are we dealing with? After a moment's thought, we realise that we are working in the pullback of the index set functor $U_{\mathsf{Set}} : \mathsf{Fam}\,\mathsf{Set} \to \mathsf{Set}$ along another index set functor $U_D : \mathsf{Fam}\,D \to \mathsf{Set}$: our objects are triples (A, B, T) where $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$ and $T : A \to D$. Hence, since D is discrete, a morphism from (A, B, T) to (A', B', T') is a pair (f, g) such that $f : A \to A'$, $g : (x : A) \to B(x) \to B'(f(x))$ and $T(x) = T'(f(x)) : D$ for each $x : A$. Assuming function extensionality, the last equation is equivalent to the equation $T = T' \circ f : A \to D$, and this is the formulation we are going to use.

Lemma 6.2 For each $\gamma : \mathsf{SP}^0_{\mathsf{IIR,A}}$, $\mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma)$ extends to a functor, i.e. given $f : X \to X'$ and $g : (x : X) \to Y(x) \to Y'(f(x))$ such that $p : Q \equiv_{A \to D} Q' \circ f$, one can define $\mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, f, g, p) : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X, Y, Q) \to \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X', Y', Q')$.

Proof. The proof is the same as the proof of Lemma 3.9, except for one complication: in the A-ind case, we need to use the proof $p : Q \equiv_{A \to D} Q' \circ f$ to make progress. We are given a pair $\langle j, y\rangle$ where $j : K \to X$ and $y : \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\text{ref}} + K, \gamma(Q \circ j), X, Y, Q, [\mathsf{rep}_{\mathsf{X}}, j])$. By composing with $f : X \to X'$, we get a first component $f \circ j : K \to X'$. By the induction hypothesis, we have a second component

$$\mathsf{Arg}_{\mathsf{IIR,A}}(\gamma(T \circ j), f, g, p, y) : \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\text{ref}} + K, \gamma(Q \circ j), X', Y', Q', [\mathsf{rep}'_{\mathsf{X}}, f \circ j]) \ ,$$

but we need something of type $\mathsf{Arg}_{\mathsf{IIR,A}}(X_{\text{ref}} + K, \gamma(Q' \circ f \circ j), X', Y', Q', [\mathsf{rep}'_{\mathsf{X}}, f \circ j])$, i.e. $\gamma(Q \circ j)$ should be $\gamma(Q' \circ f \circ j)$. For this reason, we asked for a proof $p : Q \equiv_{A \to D} Q' \circ f$, which gives rise to a proof $p' \coloneqq \mathsf{cong}(\lambda w.\, \gamma(w \circ j), p) : \gamma(Q \circ j) \equiv_{\mathsf{SP}_{\mathsf{IIR,A}}(X_{\text{ref}} + K)} \gamma(Q' \circ g \circ j)$. We can now transport the term arising from the induction hypothesis along this proof, giving

$$\mathsf{Arg}_{\mathsf{IIR,A}}(\mathsf{A\text{-}ind}(A, \gamma), f, g, p, \langle j, y\rangle) = \langle f \circ j, \mathsf{subst}(P, p', \mathsf{Arg}_{\mathsf{IIR,A}}(\gamma(T \circ j), f, g, p, y))\rangle$$

where $P(z) := \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}} + K, z, X', Y', Q', [\mathsf{rep}'_{\mathsf{X}}, f \circ j])$. The other cases are unproblematic. □

We also need a kind of coherence property of $\mathsf{Fun}_{\mathsf{IIR}}$ and $\mathsf{Arg}_{\mathsf{IIR,A}}$: given $f : X \to X'$, $g : (x : X) \to Y(x) \to Y'(f(x))$, $p : Q \equiv_{A \to D} Q' \circ f$ and $x : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X, Y, Q, \mathsf{rep}_{\mathsf{X}})$, we can either use $\mathsf{Fun}_{\mathsf{IIR}}(\gamma, X, Y, Q)$ to map x to D directly, or we can first use the functoriality of $\mathsf{Arg}^0_{\mathsf{IIR,A}}$ to first send x to $\mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, f, g, p, x) : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X', Y', Q')$ and then use $\mathsf{Fun}_{\mathsf{IIR}}(\gamma, X', Y', Q')$. The following lemma says that the result is the same:

Lemma 6.3 For each $\gamma : \mathsf{SP}^0_{\mathsf{IIR,A}}$, $f : X \to X'$, $g : (x : X) \to Y(x) \to Y'(f(x))$, $p : Q \equiv_{A \to D} Q' \circ f$ and $x : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X, Y, Q)$, there is a term

$$\mathsf{Fun}^0_{\mathsf{IIR}}\text{-}\mathsf{coh}(\gamma, x) : \mathsf{Fun}^0_{\mathsf{IIR}}(\gamma, X, Y, Q, x) \equiv_D \mathsf{Fun}^0_{\mathsf{IIR}}(\gamma, X', Y', T', \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, f, g, p, x)) \ .$$

Proof. As usual, we define a more general

$$\mathsf{Fun}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\gamma, x) : \mathsf{Fun}_{\mathsf{IIR}}(\gamma, X, Y, Q, x) \equiv_D \mathsf{Fun}_{\mathsf{IIR}}(\gamma, X', Y', T', \mathsf{Arg}_{\mathsf{IIR,A}}(\gamma, f, g, p, x))$$

for $\gamma : \mathsf{SP}_{\mathsf{IIR,A}}(X_{\mathrm{ref}})$ and $x : \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}, \gamma, X, Y, Q)$ by induction on γ. The base case $\mathsf{nil}(d)$ is trivial ($\mathsf{refl} : d \equiv_D d$), and the only case which does not follow immediately from the induction hypothesis is A-ind. After unfolding the definitions, we are looking for a term of type

$$\mathsf{Fun}_{\mathsf{IIR}}(\gamma(Q \circ j), x) \equiv_D \mathsf{Fun}_{\mathsf{IIR}}(\gamma(Q' \circ f \circ j), \mathsf{subst}(\ldots, \mathsf{Arg}_{\mathsf{IIR,A}}(\gamma(Q \circ j), f, g, p, x)))$$

By the induction hypothesis, we have a term $\mathsf{Fun}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\gamma(Q \circ j), x)$ of type

$$\mathsf{Fun}_{\mathsf{IIR}}(\gamma(Q \circ j), x) \equiv_D \mathsf{Fun}_{\mathsf{IIR}}(\gamma(Q \circ j), \mathsf{Arg}_{\mathsf{IIR,A}}(\gamma(Q \circ j), f, g, p, x))$$

and we also have a term

$$p' := \mathsf{cong}(\lambda w.\, \gamma(w \circ j), p) : \gamma(Q \circ j) \equiv_{\mathsf{SP}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}+K)} \gamma(Q' \circ g \circ j)$$

derived from $p : Q \equiv_{A \to D} Q' \circ f$. If we apply $\mathsf{cong}_2(\mathsf{Fun}_{\mathsf{IIR}})$ from Lemma 2.21 to p', we are left with the goal

$$\mathsf{subst}(\ldots, \mathsf{Arg}_{\mathsf{IIR,A}}(\gamma(Q \circ j), f, g, p, x)) \equiv \mathsf{subst}(\ldots, \mathsf{Arg}_{\mathsf{IIR,A}}(\gamma(Q \circ j), f, g, p, x))$$

which is inhabited by refl. Hence by transitivity, we are done. □

We now generalise the construction of $\mathsf{A}\text{-}\mathsf{Term}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ and $\mathsf{B}\text{-}\mathsf{Term}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}})$ from Section 3.2.3.2. It will be necessary to also include a function $\mathsf{FunA}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\ldots) : \mathsf{A}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\ldots) \to D$ which is a syntactic representation of the recursive function $T : A \to D$. To be able to define this, we must ask for functions $T_{X_{\mathrm{ref}}} : X_{\mathrm{ref}} \to D$ and $T_{Y_{\mathrm{ref}}} : Y_{\mathrm{ref}} \to D$. Thus, for $\gamma : \mathsf{SP}^0_{\mathsf{IIR,A}}$, $X_{\mathrm{ref}} : \mathsf{Set}$, $Y_{\mathrm{ref}} : \mathsf{Set}$, $T_{X_{\mathrm{ref}}} : X_{\mathrm{ref}} \to D$ and $T_{Y_{\mathrm{ref}}} : Y_{\mathrm{ref}} \to D$, we have formation rules

$$\begin{aligned}
\mathsf{A}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}) &: \mathsf{Set} \\
\mathsf{B}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}) &: \mathsf{A}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}) \to \mathsf{Set} \\
\mathsf{FunA}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}) &: \mathsf{A}\text{-}\mathsf{Term}_{\mathsf{IIR}}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}) \to D
\end{aligned}$$

The introduction rules for $\mathsf{A\text{-}Term}_{\mathsf{IIR}}$ are the same as the rules for $\mathsf{A\text{-}Term}$, except the need for $\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}$ in the $\mathsf{arg}_{\mathsf{IIR}}$ constructor:

$$\frac{x : X_{\mathrm{ref}}}{\mathsf{a}_{\mathsf{ref,IIR}}(x) : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}})}$$

$$\frac{x : Y_{\mathrm{ref}}}{\mathsf{b}_{\mathsf{ref,IIR}}(x) : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}})}$$

$$\frac{x : \mathsf{Arg}^{0}_{\mathsf{IIR,A}}(\gamma_{\mathrm{A}}, \mathsf{A\text{-}Term}(\ldots), \mathsf{B\text{-}Term}(\ldots), \mathsf{FunA\text{-}Term}_{\mathsf{IIR}}(\ldots))}{\mathsf{arg}_{\mathsf{IIR}}(x) : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}})}$$

The functions $\mathsf{B\text{-}Term}_{\mathsf{IIR}}$ and $\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}$ are defined by

$$\begin{aligned}
\mathsf{B\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \mathsf{a}_{\mathsf{ref}}(x)) &= \mathbf{0}\\
\mathsf{B\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \mathsf{b}_{\mathsf{ref}}(x)) &= \mathbf{1}\\
\mathsf{B\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \mathsf{arg}(x)) &= \mathbf{0}
\end{aligned}$$

$$\begin{aligned}
\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \mathsf{a}_{\mathsf{ref}}(x)) &= T_{X_{\mathrm{ref}}}(x)\\
\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \mathsf{b}_{\mathsf{ref}}(x)) &= T_{Y_{\mathrm{ref}}}(x)\\
\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \mathsf{arg}(x)) &= \mathsf{Fun}_{\mathsf{IIR}}(\gamma, x)
\end{aligned}$$

We move on to the interpretation of $\mathsf{A\text{-}Term}_{\mathsf{IIR}}$ and $\mathsf{B\text{-}Term}_{\mathsf{IIR}}$ by defining functions

$$\begin{aligned}
\overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots) &: \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\ldots) \to X\\
\overline{\mathsf{rep}_{\mathsf{B,IIR}}}(\ldots) &: (x : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\ldots)) \to \mathsf{B\text{-}Term}_{\mathsf{IIR}}(\ldots, x) \to Y(\overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots, x))\\
\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\ldots) &: (x : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\ldots)) \to \mathsf{FunA\text{-}Term}_{\mathsf{IIR}}(\gamma, x) \equiv_{D} Q(\overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots, x))
\end{aligned}$$

where quite a lot goes where we have written "..."; we have collected these omitted arguments in Figure 6.1. For obvious reasons, we will suppress as many of these as possible. Notice that we have included coherence proofs for Q, $T_{X_{\mathrm{ref}}}$ and $T_{Y_{\mathrm{ref}}}$.

Given all these arguments, we can define

$$\begin{aligned}
\overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots, \mathsf{rep}_{\mathsf{X}}, \ldots, \mathsf{a}_{\mathsf{ref,IIR}}(x)) &= \mathsf{rep}_{\mathsf{X}}(x)\\
\overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots, \mathsf{rep}_{\mathsf{index}}, \ldots, \mathsf{b}_{\mathsf{ref,IIR}}(x)) &= \mathsf{rep}_{\mathsf{index}}(x)\\
\overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots, T\text{-}\mathsf{coh}, T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh}, T_{Y_{\mathrm{ref}}}\text{-}\mathsf{coh}, \mathsf{arg}_{\mathsf{IIR}}(x)) &=\\
\mathsf{intro}_{\mathsf{A}}(\mathsf{Arg}^{0}_{\mathsf{IIR,A}}(\gamma_{\mathrm{A}}, \overline{\mathsf{rep}_{\mathsf{A,IIR}}}(\ldots), \overline{\mathsf{rep}_{\mathsf{B,IIR}}}(\ldots), &\mathsf{ext}(\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\ldots)), x))
\end{aligned}$$

Note how we used function extensionality ext for the $\mathsf{arg}_{\mathsf{IIR}}$ case. The simultaneously defined $\overline{\mathsf{rep}_{\mathsf{B,IIR}}}$ stays the same as $\overline{\mathsf{rep}_{\mathsf{B}}}$:

$$\begin{aligned}
\overline{\mathsf{rep}_{\mathsf{B,IIR}}}(\ldots, \mathsf{a}_{\mathsf{ref,IIR}}(x), y) &= !(y)\\
\overline{\mathsf{rep}_{\mathsf{B,IIR}}}(\ldots, \mathsf{rep}_{\mathsf{Y}}, \mathsf{b}_{\mathsf{ref,IIR}}(x), \star) &= \mathsf{rep}_{\mathsf{Y}}(y)\\
\overline{\mathsf{rep}_{\mathsf{B,IIR}}}(\ldots, \mathsf{arg}_{\mathsf{IIR}}(x), y) &= !(y)
\end{aligned}$$

$$
\begin{aligned}
\gamma_{\mathrm{A}} &: \mathsf{SP}^{0}_{\mathsf{IIR,A}}\\
X_{\mathrm{ref}} &: \mathsf{Set}\\
Y_{\mathrm{ref}} &: \mathsf{Set}\\
T_{X_{\mathrm{ref}}} &: X_{\mathrm{ref}} \to D\\
T_{Y_{\mathrm{ref}}} &: Y_{\mathrm{ref}} \to D\\
X &: \mathsf{Set}\\
Y &: X \to \mathsf{Set}\\
Q &: X \to D\\
\mathsf{intro}_{\mathrm{A}} &: \mathsf{Arg}^{0}_{\mathsf{IIR,A}}(\gamma, X, Y, Q) \to X\\
\mathsf{rep}_{\mathrm{X}} &: X_{\mathrm{ref}} \to X\\
\mathsf{rep}_{\mathrm{index}} &: Y_{\mathrm{ref}} \to X\\
\mathsf{rep}_{\mathrm{Y}} &: (b : Y_{\mathrm{ref}}) \to Y(\mathsf{rep}_{\mathrm{index}}(b))\\
T\text{-}\mathsf{coh} &: (x : \mathsf{Arg}^{0}_{\mathsf{IIR,A}}(\gamma, X, Y, Q)) \to Q(\mathsf{intro}_{\mathrm{A}}(\gamma, x)) \equiv_{D} \mathsf{Fun}^{0}_{\mathsf{IIR}}(\gamma, x)\\
T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh} &: (x : X_{\mathrm{ref}}) \to T_{X_{\mathrm{ref}}}(x) \equiv_{D} Q(\mathsf{rep}_{\mathrm{X}}(x))\\
T_{Y_{\mathrm{ref}}}\text{-}\mathsf{coh} &: (x : Y_{\mathrm{ref}}) \to T_{Y_{\mathrm{ref}}}(x) \equiv_{D} Q(\mathsf{rep}_{\mathrm{index}}(x))
\end{aligned}
$$

Figure 6.1: Omitted arguments for $\overline{\mathsf{rep}_{\mathrm{A,IIR}}}$, $\overline{\mathsf{rep}_{\mathrm{B,IIR}}}$, $\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}$, $\mathsf{Arg}_{\mathsf{IIR,B}}$ and $\mathsf{Index}_{\mathsf{IIR,B}}$.

Finally, $\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}$ is defined by case distinction on $x : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\ldots)$; if $x = \mathsf{a}_{\mathrm{ref,IIR}}(y)$ or $x = \mathsf{b}_{\mathrm{ref,IIR}}(z)$, then $T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh}(y)$ or $T_{Y_{\mathrm{ref}}}\text{-}\mathsf{coh}(z)$ respectively gives us what we need, while Lemma 6.3 together with T-coh takes care of the last case:

$$
\begin{aligned}
&\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\ldots, \mathsf{a}_{\mathrm{ref,IIR}}(y)) = T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh}(y)\\
&\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\ldots, \mathsf{b}_{\mathrm{ref,IIR}}(z)) = T_{Y_{\mathrm{ref}}}\text{-}\mathsf{coh}(z)\\
&\mathsf{FunA\text{-}Term}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\ldots, \mathsf{arg}_{\mathsf{IIR}}(w)) = \mathsf{trans}(\mathsf{Fun}_{\mathsf{IIR}}\text{-}\mathsf{coh}(\gamma, w), \mathsf{sym}(T\text{-}\mathsf{coh}(\mathsf{Arg}^{0}_{\mathsf{IIR,A}}(\ldots, w)))) \ .
\end{aligned}
$$

There is no recursion involved.

6.1.1.3 The universe $\mathsf{SP}^{0}_{\mathsf{IIR,B}}$ of descriptions of B

We introduce the universe $\mathsf{SP}_{\mathsf{IIR,B}}$ of descriptions for B. It will look a lot like SP_{B}, but we also need syntactic representations $T_{X_{\mathrm{ref}}} : X_{\mathrm{ref}} \to D$ and $T_{Y_{\mathrm{ref}}} : Y_{\mathrm{ref}} \to D$ for the value of the recursive function Q on the elements of X we know. Hence we have formation rule

$$
\frac{X_{\mathrm{ref}}, Y_{\mathrm{ref}} : \mathsf{Set} \qquad T_{X_{\mathrm{ref}}} : X_{\mathrm{ref}} \to D \qquad T_{Y_{\mathrm{ref}}} : Y_{\mathrm{ref}} \to D \qquad \gamma_{\mathrm{A}} : \mathsf{SP}_{\mathsf{IIR,A}}}{\mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}})\ \mathsf{type}}
$$

We define a $\gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathsf{IIR,A}} \vdash \mathsf{SP}^0_{\mathsf{IIR,B}}(\gamma_{\mathrm{A}})$ type by $\mathsf{SP}^0_{\mathsf{IIR,B}}(\gamma_{\mathrm{A}}) \coloneqq \mathsf{SP}_{\mathsf{IIR,B}}(\mathbf{0}, \mathbf{0}, !_D, !_D.\gamma_{\mathrm{A}})$ as usual.

The codes in $\mathsf{SP}_{\mathsf{IIR,B}}$ are very similar to the codes in SP_{B}, except we now also need to keep track of $T_{X_{\mathrm{ref}}}$ and $T_{Y_{\mathrm{ref}}}$.

$$\frac{\widehat{a} : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}})}{\mathsf{nil}(\widehat{a}) : \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}})}$$

The code $\mathsf{nil}(\widehat{a})$ represents a trivial constructor $c : \mathbf{1} \to B(a)$ (a base case), where the index a is encoded by $\widehat{a} : \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}})$.

$$\frac{K : \mathsf{Set} \qquad \gamma : K \to \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}})}{\mathsf{non\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}}))}$$

The code $\mathsf{non\text{-}ind}(K, \gamma)$ represents a non-inductive argument $x : K$, with the rest of the arguments given by $\gamma(x)$.

$$\frac{K : \mathsf{Set} \qquad \gamma : (t : K \to D) \to \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}} + K, Y_{\mathrm{ref}}, [T_{X_{\mathrm{ref}}}, t], T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}})}{\mathsf{A\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}})}$$

The code $\mathsf{A\text{-}ind}(K, \gamma)$ represents an inductive argument $j : K \to A$, with the rest of the arguments given by $\gamma(T \circ j)$. Notice how $T_{X_{\mathrm{ref}}}$ is extended.

$$\frac{\begin{array}{c} K : \mathsf{Set} \\ h_{\mathrm{index}} : K \to \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\ldots) \quad \gamma : (t : K \to D) \to \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}} + K, T_{X_{\mathrm{ref}}}, [T_{Y_{\mathrm{ref}}}, t], \gamma_{\mathrm{A}}) \end{array}}{\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma) : \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}})}$$

The code $\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma)$ represents an inductive argument with type $(x : K) \to B(i(x))$, where the index $i(x)$ is determined by h_{index}, and the rest of the arguments are given by $\gamma(T \circ i)$.

We now define the decoding function $\mathsf{Arg}_{\mathsf{IIR,B}}$. It has formation rule

$$\mathsf{Arg}_{\mathsf{IIR,B}}(\ldots) : \mathsf{SP}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, Y_{\mathrm{ref}}, T_{X_{\mathrm{ref}}}, T_{Y_{\mathrm{ref}}}, \gamma_{\mathrm{A}}) \to \mathsf{Set}$$

where "..." contains the arguments from Figure 6.1. The defining equations are the same as for $\mathsf{Arg}_{\mathsf{B}}$ from Section 3.2.3.3, except that we now also have to build up the coherence proofs $T_{X_{\mathrm{ref}}}$-coh and $T_{Y_{\mathrm{ref}}}$-coh as we go along.

$$\begin{aligned}
&\mathsf{Arg}_{\mathsf{IIR,B}}(\_\_, \mathsf{nil}(\widehat{a})) = \mathbf{1} \\
&\mathsf{Arg}_{\mathsf{IIR,B}}(\_\_, \mathsf{non\text{-}ind}(K, \gamma)) = (x : K) \times \mathsf{Arg}_{\mathsf{IIR,B}}(\_\_, \gamma(x)) \\
&\mathsf{Arg}_{\mathsf{IIR,B}}(X_{\mathrm{ref}}, \_, \_, X, \_, Q, \mathsf{rep}_{\mathrm{X}}, \_\_, T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh}, \_, \mathsf{A\text{-}ind}(K, \gamma)) \\
&\quad = (j : K \to X) \times \mathsf{Arg}_{\mathsf{IIR,B}}(X_{\mathrm{ref}} + K, \_\_, [\mathsf{rep}_{\mathrm{X}}, j], \_, \_, \_, [T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh}, (\lambda k.\, \mathsf{refl})], \_, \gamma(Q \circ j)) \\
&\mathsf{Arg}_{\mathsf{IIR,B}}(\_, Y_{\mathrm{ref}}, \gamma_{\mathrm{A}}, \_, Y, \mathsf{intro}_{\mathrm{A}}, \mathsf{rep}_{\mathrm{X}}, \mathsf{rep}_{\mathrm{index}}, \mathsf{rep}_{\mathrm{Y}}, \_\_, \mathsf{B\text{-}ind}(K, h, \gamma)) \\
&\quad = (j : (x : K) \to Y((\overline{\mathsf{rep}_{\mathrm{A}}}(\ldots) \circ h)(x))) \times \\
&\qquad\qquad \mathsf{Arg}_{\mathsf{IIR,B}}(\_, Y_{\mathrm{ref}} + K, \_\_, [\mathsf{rep}_{\mathrm{index}}, \overline{\mathsf{rep}_{\mathrm{A}}}(\ldots) \circ h], [\mathsf{rep}_{\mathrm{Y}}, j], \\
&\qquad\qquad\qquad \_, \_, [T_{Y_{\mathrm{ref}}}\text{-}\mathsf{coh}, (\lambda k.\, \mathsf{refl})], \gamma(Q \circ \overline{\mathsf{rep}_{\mathrm{A}}}(\ldots) \circ h))
\end{aligned}$$

Finally, we need to define the function $\mathsf{Index}_{\mathsf{IIR,B}}$ which picks out the index which is targeted by the constructor. The definition is exactly the same as for $\mathsf{Index}_{\mathsf{B}}$, but with the extra book keeping as in the definition of $\mathsf{Arg}_{\mathsf{IIR,B}}$. For this reason, we omit the definition here, confident that the reader will go back to Section 3.2.3.3 if necessary. As usual, we define $\mathsf{Arg}^0_{\mathsf{IIR,B}}$ and $\mathsf{Index}^0_{\mathsf{IIR,B}}$ by supplying $\mathbf{0}$ for X_{ref} and Y_{ref}, and ! for all functions with domain X_{ref} or Y_{ref}. (In particular, this means that the proof obligations

$$T_{X_{\mathrm{ref}}}\text{-}\mathsf{coh} : (x : \mathbf{0}) \to !(x) \equiv_D Q(!(x))$$
$$T_{Y_{\mathrm{ref}}}\text{-}\mathsf{coh} : (x : \mathbf{0}) \to !(x) \equiv_D Q(!(x))$$

disappear, but

$$T\text{-}\mathsf{coh} : (x : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma, X, Y, Q)) \to Q(\mathsf{intro}_{\mathsf{A}}(\gamma, x)) \equiv_D \mathsf{Fun}^0_{\mathsf{IIR}}(\gamma, x)$$

still needs to be discharged.)

6.1.1.4 Formation and introduction rules

We are now ready to give the formation and introduction rules for A, $B : A \to \mathsf{Set}$ and $T : A \to D$. They all have the common premises $\gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathsf{IIR,A}}$, $\gamma_{\mathrm{B}} : \mathsf{SP}^0_{\mathsf{IIR,B}}(\gamma_{\mathrm{A}})$, which will be omitted.

Formation rules:

$$A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : \mathsf{Set} \qquad B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : A \to \mathsf{Set} \qquad T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : A \to D$$

Introduction rule for $A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{a : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}})}{\mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(a) : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}$$

Computation rule for $T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}(\mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(a)) = \mathsf{Fun}^0_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, a)$$

Introduction rule for $B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{b : \mathsf{Arg}^0_{\mathsf{IIR,B}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}, \lambda x.\,\mathsf{refl}, \gamma_{\mathrm{B}})}{\mathsf{intro}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(b) : B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}(\mathsf{Index}^0_{\mathsf{IIR,B}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}, \lambda x.\,\mathsf{refl}, \gamma_{\mathrm{B}}))}$$

Notice how we can discharge the assumption

$$\begin{aligned} T\text{-}\mathsf{coh} : (x : \mathsf{Arg}^0_{\mathsf{IIR,A}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}})) \to \\ T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}(\mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(x)) \equiv_D \mathsf{Fun}^0_{\mathsf{IIR}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, x) \end{aligned}$$

by $\lambda x.\,\mathsf{refl}$ thanks to the computation rule for $T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$.

6.1.2 Embedding inductive-recursive and inductive-inductive definitions

Hopefully it is clear how both inductive-recursive and ordinary inductive-inductive definitions can be seen as subsystems of the system we have just defined. Inductive-inductive definitions correspond to inductive-inductive-recursive definitions where we haven chosen $D = \mathbf{1}$, and inductive-recursive definitions correspond to definitions where we never make use of the code B-ind. We make this precise by defining translations between the different universes of codes.

6.1.2.1 Embedding inductive-recursive definitions

We define a translation function $\Phi : \mathsf{IR}\ D \to \mathsf{SP}_{\mathsf{IIR}(D),\mathsf{A}}(X_{\mathrm{ref}})$ for any $X_{\mathrm{ref}} : \mathsf{Set}$.

$$\begin{aligned}
\Phi(\iota(d)) &= \mathsf{nil}(d) \\
\Phi(\sigma(A, f)) &= \mathsf{non\text{-}ind}(A, \Phi \circ f) \\
\Phi(\delta(A, F)) &= \mathsf{A\text{-}ind}(A, \Phi \circ F)
\end{aligned}$$

As a code for the inductive family B, one can for instance use one of the dummy codes $\mathsf{A\text{-}ind}(\mathbf{1}, \lambda\_.\ \mathsf{nil}(\mathsf{inr}(\mathsf{a}_{\mathrm{ref}}(\star))))$ or $\mathsf{non\text{-}ind}(\mathbf{0}, !_{\mathsf{SP}^0_{\mathsf{B}}(\Phi(\gamma))})$. The following proposition is using function extensionality, since it is proven by induction over codes, and dealing with equality of higher order objects. It should however be pointed out that no such assumptions are needed for concrete codes – we really do have a correspondence on the nose.

Proposition 6.4 The translation $\Phi : \mathsf{IR}\ D \to \mathsf{SP}_{\mathsf{IIR}(D),\mathsf{A}}(X_{\mathrm{ref}})$ is correct, i.e. for all $X_{\mathrm{ref}} : \mathsf{Set}$, $U : \mathsf{Set}$, $T : U \to D$, $\mathsf{rep}_{\mathsf{X}} : X_{\mathrm{ref}} \to U$ and $B : A \to \mathsf{Set}$, there is a term

$$\Phi\text{–}\mathsf{correct}_{\mathsf{Arg}_{\mathsf{IR}}}(\gamma) : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T) \equiv_{\mathsf{Set}} \mathsf{Arg}_{\mathsf{IIR},\mathsf{A}}(X_{\mathrm{ref}}, \Phi(\gamma), U, B, T, \mathsf{rep}_{\mathsf{X}})$$

giving rise to a function $\Phi_{\mathsf{Arg}_{\mathsf{IR}}}(\gamma) : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T) \to \mathsf{Arg}_{\mathsf{IIR},\mathsf{A}}(X_{\mathrm{ref}}, \Phi(\gamma), U, B, T, \mathsf{rep}_{\mathsf{X}})$. Furthermore, there is a term

$$\Phi\text{–}\mathsf{correct}_{\mathsf{Fun}_{\mathsf{IR}}}(\gamma, x) : \mathsf{Fun}_{\mathsf{IR}}(\gamma, U, T, x) \equiv_D \mathsf{Fun}_{\mathsf{IIR}}(X_{\mathrm{ref}}, \Phi(\gamma), U, B, T, \mathsf{rep}_{\mathsf{X}}, \Phi_{\mathsf{Arg}_{\mathsf{IR}}}(\gamma, x))$$

for each $x : \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T)$.

Proof. Both terms are straightforwardly defined by induction on γ. The function $\Phi_{\mathsf{Arg}_{\mathsf{IR}}}(\gamma)$ can be constructed as $\Phi_{\mathsf{Arg}_{\mathsf{IR}}}(\gamma) := \mathsf{subst}(\mathsf{id}, \Phi\text{–}\mathsf{correct}_{\mathsf{Arg}_{\mathsf{IR}}}(\gamma))$ (or simply by induction on γ). □

6.1.2.2 Embedding inductive-inductive definitions

The translation of inductive-inductive definitions to (degenerate) inductive-inductive-recursive definitions is a little bit more involved, as there are more concepts to translate. The main idea is to consider inductive-inductive-recursive definitions where the recursive function has codomain $D = \mathbf{1}$. By Proposition 2.23, all functions $T : A \to \mathbf{1}$ are then definitionally equal, e.g. to $\lambda x.\ \star$. We will write "_" for any function with codomain $\mathbf{1}$.

We first define a translation function $\Psi_{\mathsf{A}} : \mathsf{SP}_{\mathsf{A}}(X_{\mathrm{ref}}) \to \mathsf{SP}_{\mathsf{IIR(1),A}}(X_{\mathrm{ref}})$ for any $X_{\mathrm{ref}} : \mathsf{Set}$.

$$\begin{aligned}
\Psi_{\mathsf{A}}(\mathsf{nil}) &= \mathsf{nil}(\star) \\
\Psi_{\mathsf{A}}(\mathsf{non\text{-}ind}(K,\gamma)) &= \mathsf{non\text{-}ind}(K, \Psi_{\mathsf{A}} \circ \gamma) \\
\Psi_{\mathsf{A}}(\mathsf{A\text{-}ind}(K,\gamma)) &= \mathsf{A\text{-}ind}(K, \lambda\_.\ \Psi_{\mathsf{A}}(\gamma)) \\
\Psi_{\mathsf{A}}(\mathsf{B\text{-}ind}(K,h,\gamma)) &= \mathsf{B\text{-}ind}(K, h, \Psi_{\mathsf{A}}(\gamma))
\end{aligned}$$

In order to define the translation function for codes in $\mathsf{SP}_{\mathsf{B}}(\gamma)$, we must first prove the first translation correct:

Proposition 6.5 The translation $\Psi_{\mathsf{A}} : \mathsf{SP}_{\mathsf{A}}(X_{\mathrm{ref}}) \to \mathsf{SP}_{\mathsf{IIR(1),A}}(X_{\mathrm{ref}})$ is correct, i.e. there is a term

$$\Psi\text{-}\mathsf{correct}_{\mathsf{Arg}_{\mathsf{A}}}(\gamma) : \mathsf{Arg}_{\mathsf{A}}(X_{\mathrm{ref}}, \gamma, A, B, \mathsf{rep}_{\mathsf{X}}) \equiv_{\mathsf{Set}} \mathsf{Arg}_{\mathsf{IIR,A}}(X_{\mathrm{ref}}, \Phi(\gamma), A, B, \_, \mathsf{rep}_{\mathsf{X}})$$

which gives rise to a function

$$\Psi_{\mathsf{Arg}_{\mathsf{A}}}(\gamma, f, g) : \mathsf{Arg}^0_{\mathsf{A}}(\gamma, A, B) \to \mathsf{Arg}^0_{\mathsf{IIR,A}}(\Phi(\gamma), A', B', \_)$$

for $f : A \to A'$ and $g : (x : A) \to B(x) \to B'(f(x))$.

Proof. The proof $\Psi\text{-}\mathsf{correct}_{\mathsf{Arg}_{\mathsf{A}}}(\gamma)$ is defined by induction on γ, using function extensionality by necessity. The function $\Psi_{\mathsf{Arg}_{\mathsf{A}}}(\gamma, f, g)$ can be constructed as

$$\Psi_{\mathsf{Arg}_{\mathsf{A}}}(\gamma, f, g) := \mathsf{Arg}^0_{\mathsf{IIR,A}}(\Psi_{\mathsf{A}}(\gamma), f, g, \mathsf{refl}) \circ \mathsf{subst}(\mathsf{id}, \Psi\text{-}\mathsf{correct}_{\mathsf{Arg}_{\mathsf{A}}}(\gamma)) \ .$$

Note how we are using that $A \to \mathbf{1}$ is propositional in order to use refl as a proof that $T = T' \circ f$ for any $T : A \to D$ and $T' : A' \to D$. □

We now simultaneously define

$$\Psi_{\mathsf{A\text{-}Term}}(\gamma) : \mathsf{A\text{-}Term}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}) \to \mathsf{A\text{-}Term}_{\mathsf{IIR}}(\Psi_{\mathsf{A}}(\gamma), X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \_, \_)$$

and

$$\begin{aligned}
&\Psi_{\mathsf{B\text{-}Term}}(\gamma) : (x : \mathsf{A\text{-}Term}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}})) \to \\
&\qquad \mathsf{B\text{-}Term}(\gamma, X_{\mathrm{ref}}, Y_{\mathrm{ref}}, x) \to \mathsf{B\text{-}Term}_{\mathsf{IIR}}(\Psi_{\mathsf{A}}(\gamma), X_{\mathrm{ref}}, Y_{\mathrm{ref}}, \_, \_, \Psi_{\mathsf{A\text{-}Term}}(\gamma, x))
\end{aligned}$$

in a very straightforward way:

$$\begin{aligned}
\Psi_{\mathsf{A\text{-}Term}}(\gamma, \mathsf{a}_{\mathsf{ref}}(x)) &= \mathsf{a}_{\mathsf{ref,IIR}}(x) \\
\Psi_{\mathsf{A\text{-}Term}}(\gamma, \mathsf{b}_{\mathsf{ref}}(x)) &= \mathsf{b}_{\mathsf{ref,IIR}}(x) \\
\Psi_{\mathsf{A\text{-}Term}}(\gamma, \mathsf{arg}(x)) &= \mathsf{arg}_{\mathsf{IIR}}(\Psi_{\mathsf{Arg}_{\mathsf{A}}}(\gamma, \Psi_{\mathsf{A\text{-}Term}}(\gamma), \Psi_{\mathsf{B\text{-}Term}}(\gamma), x))
\end{aligned}$$

and

$$\begin{aligned}\Psi_{\text{B-Term}}(\gamma, \mathsf{a}_{\text{ref}}(x), y) &= y\\ \Psi_{\text{B-Term}}(\gamma, \mathsf{b}_{\text{ref}}(x), y) &= y\\ \Psi_{\text{B-Term}}(\gamma, \mathsf{arg}(x), y) &= y\end{aligned}$$

(we need to do the case distinction, as B-Term$(\ldots)$ and B-Term$_{\text{IIR}}(\ldots)$ are not equal for neutral terms). Notice how the definition of $\Psi_{\text{A-Term}}$ is making use of the correctness proof $\Psi\text{-correct}_{\text{Arg}_{\text{A}}}$ (via $\Psi_{\text{Arg}_{\text{A}}}$).

Finally, we can define $\Psi_{\text{B}} : \mathsf{SP}_{\text{B}}(X_{\text{ref}}, Y_{\text{ref}}, \gamma_{\text{A}}) \to \mathsf{SP}_{\text{IIR(1)},\text{B}}(X_{\text{ref}}, Y_{\text{ref}}, \_, \_, \Psi_{\text{B}}(\gamma_{\text{A}}))$ for any $X_{\text{ref}}, Y_{\text{ref}} : \mathsf{Set}$ and $\gamma_{\text{A}} : \mathsf{SP}^0_{\text{A}}$.

$$\begin{aligned}\Psi_{\text{B}}(\mathsf{nil}(a)) &= \mathsf{nil}(\Psi_{\text{A-Term}}(\gamma_{\text{A}}, a))\\ \Psi_{\text{B}}(\mathsf{non\text{-}ind}(K, \gamma)) &= \mathsf{non\text{-}ind}(K, \Psi_{\text{B}} \circ \gamma)\\ \Psi_{\text{B}}(\mathsf{A\text{-}ind}(K, \gamma)) &= \mathsf{A\text{-}ind}(K, \lambda\_.\, \Psi_{\text{B}}(\gamma))\\ \Psi_{\text{B}}(\mathsf{B\text{-}ind}(K, h, \gamma)) &= \mathsf{B\text{-}ind}(K, \Psi_{\text{A-Term}}(\gamma_{\text{A}}) \circ h, \lambda\_.\, \Psi_{\text{B}}(\gamma))\end{aligned}$$

We can also prove that the translation Ψ_{B} is correct, after proving that $\overline{\mathsf{rep}_{\text{A,IIR}}}$ and $\overline{\mathsf{rep}_{\text{A}}}$ agree, but as this is not very enlightening, we shall skip doing so.

6.1.3 Extending the model

We now recall Dybjer and Setzer's [1999] model construction for inductive-recursive definitions, with the goal of merging it with the model from Section 5.1 to cover inductive-inductive-recursive definitions as well. As we will see, this will not present much additional difficulty.

6.1.3.1 Dybjer and Setzer's model for inductive-recursive definitions

Dybjer and Setzer [1999] constructs a set-theoretical model for inductive-recursive definitions. The Logical Framework and the standard type formers are interpreted as in Section 5.1. The inductive-recursive definition described by code $\gamma : \mathsf{IR}\,(D)$ is modelled as the result of iterating $(\mathsf{Arg}_{\text{IR}}, \mathsf{Fun}_{\text{IR}})$ a suitably large number of times as in the sequence

$$0 \subseteq \mathsf{Arg}_{\text{IR}}(\gamma, 0, !_D) \subseteq \mathsf{Arg}_{\text{IR}}(\gamma, \mathsf{Arg}_{\text{IR}}(\gamma, 0, !_D), \mathsf{Fun}_{\text{IR}}(\gamma, 0, !_D)) \subseteq \ldots$$

with corresponding decoding functions

$$!_D, \qquad \mathsf{Fun}_{\text{IR}}(\gamma, 0, !_D), \qquad \mathsf{Arg}_{\text{IR}}(\gamma, \mathsf{Arg}_{\text{IR}}(\gamma, 0, !_D), \mathsf{Fun}_{\text{IR}}(\gamma, 0, !_D)), \qquad \ldots$$

How large is suitably large? Dybjer and Setzer assume that a Mahlo cardinal M exists, and do M iterations.

Definition 6.6 (Mahlo) A cardinal κ is a (strong) *Mahlo cardinal* if it is inaccessible and every normal function $f : \kappa \to \kappa$ has an inaccessible fixed point. ◼

Recall that a function f is normal if it is strictly increasing and continuous at limits (i.e. $f(\lambda) = \sup_{\beta<\lambda} f(\beta)$). A Mahlo cardinal is large in the technical sense: ZFC cannot prove its existence[1], since if M is a Mahlo cardinal, then V_{M} is a model of ZFC. Hence, if ZFC could prove that a Mahlo cardinal exists, it would prove its own consistency, which contradicts Gödel's second incompleteness theorem. Hence, the consistency proof is necessarily a relative one, but this is of course always the case.

The constant Set is interpreted as $[\![\mathsf{Set}]\!] := V_{\mathsf{M}}$, and as hinted above, we define $[\![U_\gamma]\!] := U_\gamma^{\mathsf{M}}$ and $[\![T_\gamma]\!](x) := T_\gamma^{\mathsf{M}}(x)$ where

$$U_\gamma^\alpha = \mathsf{Arg}_{\mathsf{IR}}(\gamma, \bigcup_{\beta<\alpha} U_\gamma^\beta, \bigcup_{\beta<\alpha} T_\gamma^\beta) \qquad\qquad T_\gamma^\alpha = \mathsf{Fun}_{\mathsf{IR}}(\gamma, \bigcup_{\beta<\alpha} U_\gamma^\beta, \bigcup_{\beta<\alpha} T_\gamma^\beta)$$

The tricky part is to show that this sequence converges after κ steps for some $\kappa < \mathsf{M}$, so that $[\![U_\gamma]\!] \in [\![\mathsf{Set}]\!] = V_{\mathsf{M}}$. The proof proceeds in two stages; First, we prove that if we can find an inaccessible bound κ for cardinality of the premises (index sets) of inductive arguments as we iterate the initial sequence, then we will find a fixed point after κ iterations [Dybjer and Setzer, 1999, Lemma 2]. This proof does not use anything beyond ZFC (and the assumption that the inaccessible bound exists). Using the Mahlo property, we then find such a bound [Dybjer and Setzer, 1999, Lemma 3], and can then conclude the proof using the standard argument. Whenever we mention sets or families in the rest of this section, we mean sets inside the model, i.e. elements of V_{M}.

Before we can start, we need some more technical facts. Recall that the rank $\mathrm{rk}(x)$ of x is the least α such that $x \in V_{\alpha+1}$.

Lemma 6.7

(i) $V_\alpha = \{x \mid \mathrm{rk}(x) < \alpha\}$.

(ii) If $x \in y$ then $\mathrm{rk}(x) < \mathrm{rk}(y)$, and if $x \subseteq y$, then $\mathrm{rk}(x) \leq \mathrm{rk}(y)$.

(iii) For all y, $\mathrm{rk}(y) = \sup\{\mathrm{rk}(x) + 1 \mid x \in y\}$.

(iv) Let κ be regular. If $A \in V_\kappa$ then $\bigcup A \in V_\kappa$. □

The functor $[\![\gamma]\!] := (\mathsf{Arg}_{\mathsf{IR}}(\gamma), \mathsf{Fun}_{\mathsf{IR}}(\gamma))$ is monotone in the following sense:

Lemma 6.8 Let φ be an IR code and (U,T), (U',T') objects of $\mathsf{Fam}\,|\mathbb{C}|$. Assume $U \subseteq U'$ and $T' \restriction U = T$. Then

(i) $\mathsf{Arg}_{\mathsf{IR}}(\varphi, U, T) \subseteq \mathsf{Arg}_{\mathsf{IR}}(\varphi, U', T')$, and

(ii) $\mathsf{Fun}_{\mathsf{IR}}(\varphi, U', T') \restriction \mathsf{Arg}_{\mathsf{IR}}(\varphi, U, T) = \mathsf{Fun}_{\mathsf{IR}}(\varphi, U, T)$. □

[1] Assuming that ZFC is consistent, of course.

Definition 6.9 Given an IR code φ and an object (U, T) of $\mathsf{Fam}\,|\mathbb{C}|$, the set $\mathsf{Aux}(\varphi, U, T) \subseteq V_{\mathsf{M}}$ of premises of inductive arguments of ϕ with respect to U, T is defined by induction over φ:

$$\begin{aligned}
\mathsf{Aux}(\iota(d), U, T) &= \varnothing \\
\mathsf{Aux}(\sigma(A, \varphi), U, T) &= \bigcup_{x \in A} \mathsf{Aux}(\varphi(x), U, T) \\
\mathsf{Aux}(\delta(A, \varphi), U, T) &= \{A\} \cup \bigcup_{f : A \to U} \mathsf{Aux}(\varphi(T \circ f), U, T)
\end{aligned}$$ ■

Remark 6.10 We define Aux differently compared to Dybjer and Setzer [1999]: We collect all the sets we are interested in, whereas they use products and coproducts to build one big set that "contain" all interesting sets.

Lemma 6.11 Let κ be inaccessible and $(U^\alpha, T^\alpha)_{\alpha<\kappa}$ be a monotone κ-sequence of objects of $\mathsf{Fam}\,|\mathbb{C}|$, i.e. if $\alpha < \beta$ then $U^\alpha \subseteq U^\beta$ and $T^\beta \restriction U^\alpha = T^\alpha$. Assume for some $\alpha_0 < \kappa$ that

$$\mathsf{Aux}(\varphi, U^\alpha, T^\alpha) \subseteq V_\kappa \tag{6.1}$$

for all $\alpha_0 \leq \alpha < \kappa$. Then $\mathsf{Arg}_{\mathsf{IR}}(\varphi, U, T)$ is κ-continuous in (U, T), i.e.

$$\mathsf{Arg}_{\mathsf{IR}}(\varphi, \bigcup_{\alpha<\kappa} U^\alpha, \bigcup_{\alpha<\kappa} T^\alpha) = \bigcup_{\alpha<\kappa} \mathsf{Arg}_{\mathsf{IR}}(\varphi, U^\alpha, T^\alpha) \ .$$

Notation. For readability, let us write $\bigcup_{\alpha<\kappa}(U^\alpha, T^\alpha)$ for $(\bigcup_{\alpha<\kappa} U^\alpha, \bigcup_{\alpha<\kappa} T^\alpha)$.

Proof. The direction $\supseteq$ follows immediately from Lemma 6.8. We prove $\subseteq$ by induction over φ:

- If $\varphi = \iota(d)$, then

$$\mathsf{Arg}_{\mathsf{IR}}(\iota(d), \bigcup_{\alpha<\kappa}(U^\alpha, T^\alpha) = \mathbf{1} = \bigcup_{\alpha<\kappa} \mathbf{1} = \bigcup_{\alpha<\kappa} \mathsf{Arg}_{\mathsf{IR}}(\iota(d), U^\alpha, T^\alpha) \ .$$

- If $\varphi = \sigma(A, \varphi')$, then first note that since $\mathsf{Aux}(\sigma(A, \varphi'), U^\alpha, T^\alpha) \subseteq V_\kappa$, we have $\mathsf{Aux}(\varphi'(x), U^\alpha, T^\alpha) \subseteq V_\kappa$ for all $\alpha_0 \leq \alpha < \kappa$ and $x \in A$ by the definition of Aux. Now

$$\begin{aligned}
\mathsf{Arg}_{\mathsf{IR}}(\sigma(A, \varphi'), \bigcup_{\alpha<\kappa}(U^\alpha, T^\alpha) &= \Sigma_{x \in A} \mathsf{Arg}_{\mathsf{IR}}(\varphi'(x), \bigcup_{\alpha<\kappa}(U^\alpha, T^\alpha) \\
&\subseteq \Sigma_{x \in A} \bigcup_{\alpha<\kappa} \mathsf{Arg}_{\mathsf{IR}}(\varphi'(x), U^\alpha, T^\alpha) \\
&= \bigcup_{\alpha<\kappa} \Sigma_{x \in A} \mathsf{Arg}_{\mathsf{IR}}(\varphi'(x), U^\alpha, T^\alpha) \\
&= \bigcup_{\alpha<\kappa} \mathsf{Arg}_{\mathsf{IR}}(\sigma(A, \varphi'), U^\alpha, T^\alpha)
\end{aligned}$$

  where the inclusion follows from the induction hypothesis.

- If $\varphi = \delta(A, \varphi')$, then assume $a \in \mathsf{Arg}_{\mathsf{IR}}(\delta(A,\varphi'), \bigcup_{\alpha<\kappa} U^\alpha, \bigcup_{\alpha<\kappa} T^\alpha)$. We want to find $\alpha < \kappa$ such that $a \in \mathsf{Arg}_{\mathsf{IR}}(\delta(A,\varphi'), U^\alpha, T^\alpha)$. We know $a = \langle f, y\rangle$ for some $f : A \to \bigcup_{\alpha<\kappa} U^\alpha$ and
$$y \in \mathsf{Arg}_{\mathsf{IR}}(\varphi'(\bigcup_{\alpha<\kappa} T^\alpha \circ f), \bigcup_{\alpha<\kappa}(U^\alpha, T^\alpha).$$
<u>**Claim:**</u> $f : A \to \bigcup_{\alpha<\beta} U^\alpha$ for some $\beta < \kappa$.
Proof of claim. Assume not, i.e. for all $\beta < \kappa$ there exists $x \in A$ such that $f(x) \notin \bigcup_{\alpha<\beta} U^\alpha$. By (6.1), we have $A \in V_\kappa$, i.e. $\mathrm{rk}(A) < \kappa$ by Lemma 6.7(i). We define a strictly increasing function $g : \mathrm{rk}(A) \to \kappa$ by transfinite recursion:
$$g(\gamma) = \min\{\beta \mid (\forall\gamma' < \gamma)(g(\gamma') < \beta) \wedge f[A \cap V_\gamma] \subseteq \bigcup_{\alpha<\beta} U^\alpha\}$$
By assumption, $\sup g = \kappa$ which contradicts the regularity of κ. Hence $f : A \to \bigcup_{\alpha<\beta} U^\alpha$ for some $\beta < \kappa$. □
Since $\bigcup_{\alpha<\beta} U^\alpha \subseteq U^\beta$, we particularly have $f : A \to U^\beta$, and w.l.o.g. we can assume $\alpha_0 \leq \beta$ (if not, choose $\beta' := \alpha_0$; we still have $f : A \to U^{\beta'}$ since $\beta < \beta'$, hence $U^\beta \subseteq U^{\beta'}$). For $\beta \leq \alpha < \kappa$ we have $f : A \to U^\alpha$ and thus
$$\mathsf{Aux}(\varphi'(T^\alpha \circ f), U^\alpha, T^\alpha) \subseteq V_\kappa$$
by (6.1). Hence by the induction hypothesis, there exists α' such that
$$y \in \mathsf{Arg}_{\mathsf{IR}}(\varphi'(\bigcup_{\alpha<\kappa} T^\alpha \circ f), U^{\alpha'}, T^{\alpha'}) = \mathsf{Arg}_{\mathsf{IR}}(\varphi'(T^{\alpha'} \circ f), U^{\alpha'}, T^{\alpha'})$$
and with $\alpha = \max\{\beta, \alpha'\}$, we have $a = \langle f, y\rangle \in \mathsf{Arg}_{\mathsf{IR}}(\delta(A,\varphi'), U^\alpha, T^\alpha)$ as required.

□

The proof of the following lemma is the only place where we use the existence of a Mahlo cardinal:

Lemma 6.12 Let φ be an IR code and $(U^\alpha, T^\alpha)_\alpha$ the initial sequence of the associated functor. There exists an inaccessible κ such that
$$\mathsf{Aux}(\varphi, U^\alpha, T^\alpha) \subseteq V_\kappa$$
for all $\alpha < \kappa$.

Proof. The strategy for the proof is as follows: we define an increasing function $f : \mathsf{Ord} \to \mathsf{Ord}$, which tells you how much further up the cumulative hierarchy you need to go to contain one iteration of $\mathsf{Arg}_{\mathsf{IR}}(\gamma)$. The important property of f will be
$$\text{if } U^{\beta'} \subseteq V_\beta \text{ then } U^{\beta'+1} \cup \mathsf{Aux}(\gamma, U^{\beta'}, T^{\beta'}) \subseteq V_{f(\beta)} \tag{6.2}$$
for all $\beta' < \mathsf{M}$. We then show that $f : \mathsf{M} \to \mathsf{M}$ and use the Mahlo property to find a fixed point κ of f. Finally we show $\mathsf{Aux}(\varphi, U^\alpha, T^\alpha) \subseteq V_\kappa$ by induction on α.

The function $f : \mathsf{Ord} \to \mathsf{Ord}$ is defined by transfinite recursion:

$$\begin{aligned} f(\beta) = \min\{\alpha \,|(\forall \beta' < \beta)\big(f(\beta') < \alpha\big) \wedge \\ (\forall \beta' < \mathsf{M})\big(U^{\beta'} \subseteq V_\beta \implies U^{\beta'+1} \cup \mathsf{Aux}(\gamma, U^{\beta'}, T^{\beta'}) \subseteq V_\alpha\big)\} \end{aligned}$$

The first conjunct makes sure that f is increasing, and the second makes (6.2) true.
Claim: $f : \mathsf{M} \to \mathsf{M}$.
Proof of claim. Let $\beta < \mathsf{M}$ and note that

$$\begin{aligned} f(\beta) = \min\{\alpha \,|(\forall \beta' < \beta)\big(f(\beta') < \alpha\big) \wedge \\ (\forall \beta' \in \{\beta' \in \mathsf{M} \mid U^{\beta'} \subseteq V_\beta\})\big(U^{\beta'+1} \cup \mathsf{Aux}(\gamma, U^{\beta'}, T^{\beta'}) \subseteq V_\alpha\big)\} \ , \end{aligned}$$

further that $B := \{\beta' \in \mathsf{M} \mid U^{\beta'} \subseteq V_\beta\} \in V_{\beta+1} \subseteq V_{\mathsf{M}}$ so that $|B| < \mathsf{M}$ by Proposition 5.10(iv). For each $\beta' \in B$, we have $U^{\beta'+1} \cup \mathsf{Aux}(\gamma, U^{\beta'}, T^{\beta'}) \subseteq V_{\mathsf{M}}$ and hence $U^{\beta'+1} \cup \mathsf{Aux}(\gamma, U^{\beta'}, T^{\beta'}) \subseteq V_{\alpha_{\beta'}}$ for some $\alpha_{\beta'} < \mathsf{M}$ since M is a limit. Thus $f(\beta) \leq \sup_{\beta'} \alpha_{\beta'} < \mathsf{M}$ by the regularity of M. □

So f is an increasing function on M, however f need not be continuous at limits, hence not normal and the Mahlo property might not apply. To fix this, we define a new function θ: let for $\alpha < \mathsf{M}$ $\theta(\alpha) = f^\alpha(0)$.
Claim: $\theta : \mathsf{M} \to \mathsf{M}$, and θ is normal.
Proof of claim. We prove that $\theta(\alpha) < \mathsf{M}$ for $\alpha < \mathsf{M}$ by transfinite induction over α. The base case and successor case are clear, since $f : \mathsf{M} \to \mathsf{M}$. If $\lambda < \mathsf{M}$ is a limit, then $\theta : \lambda \to \mathsf{M}$ is a normal function so that $\theta(\lambda) = \sup_{\beta<\lambda} \theta(\beta) < \mathsf{M}$ by the regularity of M. Finally θ is increasing since f is, and continuous at limits by definition. □

Hence by the Mahlo property, θ has an inaccessible fixed point $\kappa < \mathsf{M}$.
Claim: $f : \kappa \to \kappa$.
Proof of claim. Assume $\alpha < \kappa$. Since κ is a limit, $\alpha < \beta$ for some $\beta < \kappa$, and $\beta \leq \theta(\beta)$ since θ is increasing. Thus

$$f(\alpha) < f(\beta) \leq f(\theta(\beta)) = \theta(\beta + 1) < \theta(\kappa) = \kappa$$

i.e. $f : \kappa \to \kappa$. □

This combined with (6.2) gives us a very useful fact:

$$\text{if } U^{\beta'} \subseteq V_\beta \text{ then } U^{\beta'+1} \cup \mathsf{Aux}(\gamma, U^{\beta'}, T^{\beta'}) \subseteq V_\kappa \tag{6.2'}$$

for all $\beta < \kappa$ (since $f(\beta) < \kappa$, hence $V_{f(\beta)} \subseteq V_\kappa$).

Finally, we prove that $U^\alpha \subseteq V_\kappa$ for all $\alpha < \kappa$ by induction on α. By (6.2′), it then immediately follows that $\mathsf{Aux}(\varphi, U^\alpha, T^\alpha) \subseteq V_\kappa$.

- If $\alpha = 0$, then $U^0 = \varnothing \subseteq V_\kappa$.
- If $\alpha = \beta + 1$, then $U^\beta \subseteq V_\kappa$ by the induction hypothesis, and we are done by (6.2′).
- If $\alpha = \lambda$ limit, then $U^\lambda = \bigcup_{\beta<\lambda} U^\beta \subseteq V_\kappa$ by the induction hypothesis. □

Remark 6.13 At a glance, it might seem that the proof is independent of the particular definition of U^α and $\mathsf{Aux}(\varphi, U^\alpha, T^\alpha)$. However, we are making use of the particular definition of U^0 and U^λ, and that $\mathsf{Aux}(\varphi, U^\alpha, T^\alpha \subseteq V_{\mathsf{M}}$.

Theorem 6.14 Let φ be an IR code. Then $[\![\varphi]\!]$ has an initial algebra.

Proof. Feeding Lemma 6.8 and Lemma 6.12 into Lemma 6.11, we get that

$$\begin{aligned}\mathsf{Arg}_{\mathsf{IR}}(\varphi, \bigcup_{\alpha<\kappa} U^\alpha, \bigcup_{\alpha<\kappa} T^\alpha) &= \bigcup_{\alpha<\kappa} \mathsf{Arg}_{\mathsf{IR}}(\varphi, U^\alpha, T^\alpha)\\ &= \bigcup_{\alpha<\kappa} U^{\alpha+1}\\ &= \bigcup_{\alpha<\kappa} U^\alpha \;.\end{aligned}$$

By Lemma 6.8, $\mathsf{Fun}_{\mathsf{IR}}(\varphi, \bigcup_{\alpha<\kappa} U^\alpha, \bigcup_{\alpha<\kappa} T^\alpha) = \bigcup_{\alpha<\kappa} T^\alpha$, so that the initial sequence converges after κ steps. By Adámek et al. [2010, Thm 3.1.4], $[\![\varphi]\!]$ has an initial algebra. □

6.1.3.2 A model for inductive-inductive-recursive definitions

We can now reap the benefits of Dybjer and Setzer's labour, and combine their model with our from Section 5.1. As mentioned in Remark 6.13, there are not many things that need to be rechecked when we try to deploy the proof in the new setting. The proof of Theorem 6.14 is the same, mutatis mutandis. This shows that the judgements $A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : \mathsf{Set}$ and $T_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} \to \mathsf{Set}$ indeed are soundly interpreted in the model. Finally, $B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} \to \mathsf{Set}$ is taken care of as before by (a slight variation of) Theorem 5.7.

Theorem 6.15 There is a model of the theory of inductive-inductive-recursive definitions that can be constructed using ZFC and the existence of a Mahlo cardinal. □

6.2 Telescopic inductive definitions and generalised families

In this section, we extend the theory to accommodate some more liberal uses of inductive-inductive like definitions. We saw the need for this already in Section 3.1, where the second set B often was indexed not only by A, but by e.g. $A \times A$ or $\mathbb{N} \times A$. We cover such generalised families in Section 6.2.1. In Section 6.2.2, we explore another extension which is needed for Danielsson's [2007] and Chapman's [2009] formalisations of Type Theory inside Type Theory: an inductive-inductive definition of not just A and B, but of a whole telescope

$$\begin{aligned}&A : \mathsf{Set},\\ &B : A \to \mathsf{Set},\\ &C : (x : A) \to B(x) \to \mathsf{Set}\\ &\quad\vdots\end{aligned}$$

6.2.1 Generalised families

In this section, we seek to replace families of the form

$$A : \mathsf{Set}\ , \qquad B : A \to \mathsf{Set}$$

with more general families

$$A : \mathsf{Set}\ , \qquad B : F(A) \to \mathsf{Set} \tag{6.3}$$

for some type former $F : \mathsf{Set} \to \mathsf{Set}$, for instance $F(X) = X \times X$ (Example 3.3) or $F(X) = \mathbb{N} \times X$ (Example 3.2). As is often the case, it will not be enough that F only acts on types, instead we need $F : \mathsf{Set} \to \mathsf{Set}$ to be a functor. In this situation, pairs (A, B) as in (6.3) naturally form a category:

Definition 6.16 Let $F : \mathsf{Set} \to \mathsf{Set}$ be a functor. The category $\mathsf{Fam}_F(\mathbb{C})$ has objects pairs (A, B) where $A : \mathsf{Set}$ and $B : F(A) \to \mathbb{C}$. A morphism from (A, B) to (A', B') is a pair (f, g) where $f : A \to A'$ is a function and $g : B \to B' \circ F(f)$ is a natural transformation, i.e. $g : (x : F(A)) \to B(x) \to B'(F(f)(x))$. ■

We recover $\mathsf{Fam}\,\mathbb{C}$ as $\mathsf{Fam}_{\mathsf{Id}}(\mathbb{C})$. Note that it is crucial that F preserves identities and composition to be able to define them in $\mathsf{Fam}_F(\mathbb{C})$. Alternatively, $\mathsf{Fam}_F(\mathbb{C})$ can be described as the lax comma category $F \downarrow K_{\mathbb{C}}$ where $K_{\mathbb{C}} : \mathsf{Set} \to \mathsf{Cat}$ is the constantly $\mathbb{C}$-valued functor and the set $F(X)$ is considered as a discrete category. It is not clear if this is helpful for our purposes, and so, we will not develop this view further.

The goal is now to show that $\mathsf{Fam}_F(\mathsf{Set})$ is a Category with Families for well-behaved functors F, so that the development in Chapter 4 can be repeated, but with $\mathsf{Fam}_F(\mathsf{Set})$ in the place of $\mathsf{Fam}\,\mathsf{Set}$. By Theorem 4.28, it is enough to show that $\mathsf{Fam}_F(\mathsf{Set})$ has finite limits, which it does if F preserves pullbacks:

Lemma 6.17 If $F : \mathsf{Set} \to \mathsf{Set}$ preserves pullbacks and $\mathbb{C}$ has finite limits, then also $\mathsf{Fam}_F(\mathbb{C})$ has finite limits.

Proof. We show that $\mathsf{Fam}_F(\mathbb{C})$ has a terminal object and pullbacks. The terminal object in $\mathsf{Fam}_F(\mathbb{C})$ is $\mathbf{1} = (\mathbf{1}_{\mathsf{Set}}, \lambda\_.\,\mathbf{1}_{\mathbb{C}})$. The pullback of $(f, g) : (A, B) \to (C, D)$ and $(f', g') : (A', B') \to (C, D)$ is $(A \times_C A', B'')$ where $A \times_C A'$ is the pullback of f and f', and B'' can be defined since F preserves pullbacks: it is enough to define $B''(x)$ for $x : F(A) \times_{F(C)} F(A') \cong F(A \times_C A')$, but we can assume that this is the standard pullback of sets, i.e. $x = \langle y, z\rangle$ with $F(f)(y) = F(f')(z)$. Thus we can define $B''(\langle y, z\rangle) = B(y) \times_{D(F(f)(y))} B'(z)$. The unique mediating morphism is inherited from the pullbacks in Set and $\mathbb{C}$. □

Note that Proposition 4.35 (and its proof) is generalised by this lemma. Let us quickly check that the functors $F : \mathsf{Set} \to \mathsf{Set}$ involved in Examples 3.2 and 3.3 satisfy the condition of preserving pullbacks. In Example 3.3, the functor in question was defined by $F(X) = X \times X$. It can easily be checked that F is right adjoint to the functor G defined by $G(Y) = Y + Y$, hence since right adjoints preserve limits, F preserves pullbacks. In Example 3.2, we are instead using the functor defined by $F'(X) = \mathbb{N} \times X$, which cannot be a right adjoint, since it does not preserve terminal objects. It does, however, preserve pullbacks: Let $f : A \to C$ and $g : B \to C$ be given and consider their pullback object

$$A \times_C B = \{\langle x, y\rangle : A \times B \mid f(x) = g(y)\} \ .$$

Recall that $F(f) = \mathsf{id} \times f : \mathbb{N} \times A \to \mathbb{N} \times C$, so

$$\begin{aligned} F(A) \times_{F(C)} F(B) &= \{\langle\langle n, x\rangle, \langle m, y\rangle\rangle : (\mathbb{N} \times A) \times (\mathbb{N} \times B) \mid \langle n, f(x)\rangle = \langle m, g(y)\rangle\} \\ &\cong \{\langle n, x, y\rangle : \mathbb{N} \times A \times B \mid \langle n, f(x)\rangle = \langle n, g(y)\rangle\} \\ &\cong \mathbb{N} \times (A \times_C B) = F(A \times_C B) \ . \end{aligned}$$

6.2.1.1 Inductive-inductive definitions of generalised families

By Theorem 4.28, we know that $\mathsf{Fam}_F(\mathsf{Set})$ is a Category with Families, and inspecting the construction of finite limits in $\mathsf{Fam}_F(C)$, we see that the index set functor $U : \mathsf{Fam}_F(\mathbb{C}) \to \mathsf{Set}$ defined by $U(A, B) = A)$ preserves finite limits. Hence, by Theorem 4.37 also $\mathsf{Dialg}(\mathsf{Arg_A}, U)$ is a Category with Families for any functor $\mathsf{Arg_A} : \mathsf{Fam}_F(\mathsf{Set}) \to \mathsf{Set}$, where we of course should think of $\mathsf{Arg_A}$ as a suitable strictly positive functor which describes the domain of the constructor for the first set A. The rest of the construction from Section 4.1.2 is exactly the same, and we end up with a generalised notion of an inductive-inductive definition of $A : \mathsf{Set}$, $B : F(A) \to \mathsf{Set}$ for each pullback-preserving functor $F : \mathsf{Set} \to \mathsf{Set}$, which can be represented by two functors

$$\mathsf{Arg_A} : \mathsf{Fam}_F(\mathsf{Set}) \to \mathsf{Set} \qquad\qquad \mathsf{Arg} : \mathsf{Dialg}(\mathsf{Arg_A}, U) \to \mathsf{Fam}_F(\mathbb{C})$$

with $U \circ \mathsf{Arg} = \mathsf{Arg_A} \circ V$, where $U : \mathsf{Fam}_F(\mathbb{C}) \to \mathsf{Set}$ is the index set functor and $V : \mathsf{Dialg}(\mathsf{Arg_A}, U) \to \mathsf{Fam}_F(\mathbb{C})$ is the forgetful functor $V(A, f) = A$.

Each such pair of functors give rise to a Category with Families $\mathbb{E}_{\mathsf{Arg}}$, and Theorems 4.41 and 4.42 apply: if this category has an initial object, we have a reasonable notion of elimination rules, and if we have elimination rules, then $\mathbb{E}_{\mathsf{Arg}}$ has a weakly initial object. If we were to design a syntactical system of codes for such functors, like in Chapter 3, it seems reasonable to think that we could also get a strongly initial object from the elimination rules like in Theorem 4.43.

6.2.2 Towers of inductive-inductive definitions

We now extend the theory in another direction, namely to the simultaneous definition of a whole telescope (of fixed length) of sets. Let us first see that this is useful by considering an extension of Example 3.1, closer to what is actually presented in Danielsson [2007]:

Example 6.18 We extend the contexts and types from Example 3.1 with terms as well. The contexts stay the same:

$$\frac{}{\varepsilon : \mathsf{Ctxt}} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma)}{\Gamma \rhd \sigma : \mathsf{Ctxt}}$$

For types, we keep the base type ι and function types $\Pi_\Gamma(\sigma,\tau)$, but anticipating the definition of the terms, we also add a weakening operation: if τ is a type in context Γ, then τ should also be a type in context $\Gamma \rhd \sigma$ for all types σ. We encode this with a constructor $\mathsf{Wk}_{\Gamma,\sigma}(\tau)$:

$$\frac{\Gamma : \mathsf{Ctxt}}{\iota_\Gamma : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma) \quad \tau : \mathsf{Ty}(\Gamma \rhd \sigma)}{\Pi_\Gamma(\sigma,\tau) : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma) \quad \tau : \mathsf{Ty}(\Gamma)}{\mathsf{Wk}_{\Gamma,\sigma}(\tau) : \mathsf{Ty}(\Gamma \rhd \sigma)}$$

To see that the definition of terms can also depend on the definition of types, so that an inductive-inductive definition is really necessary, we could also consider to add a universe set with a decoding function el. We should then say that for each term t of type set, $\mathsf{el}(t)$ is a type, which we can do in the following way:

$$\frac{\Gamma : \mathsf{Ctxt}}{\mathsf{set}_\Gamma : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad t : \mathsf{Tm}(\Gamma, \mathsf{set}_\Gamma)}{\mathsf{el}_\Gamma(t) : \mathsf{Ty}(\Gamma)}$$

However, this would require yet another extension (with no particular technical difficulties), namely that later constructors for the same set can depend on earlier constructors, and so we leave out the universe and its decoding from our example.

Finally, we introduce some terms. First, we always have a term in context $\Gamma, x : A$, namely the variable x. In our nameless de Bruijn representation, we will write top for this variable. We also add a weakening wk of terms as well, so that we can reach variables further into our context, and not just the outermost one. Finally, we add lambda abstractions, written $\mathsf{lam}(t)$.

$$\frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma)}{\mathsf{top}_{\Gamma,\sigma} : \mathsf{Tm}(\Gamma \rhd \sigma, \mathsf{Wk}_{\Gamma,\sigma}(\sigma))} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma) \quad \tau : \mathsf{Ty}(\Gamma) \quad t : \mathsf{Tm}(\Gamma,\tau)}{\mathsf{wk}_{\Gamma,\sigma,\tau}(t) : \mathsf{Tm}(\Gamma \rhd \sigma, \mathsf{Wk}_{\Gamma,\sigma}(\tau))}$$

$$\frac{\Gamma : \mathsf{Ctxt} \quad \sigma : \mathsf{Ty}(\Gamma) \quad \tau : \mathsf{Ty}(\Gamma \rhd \sigma) \quad t : \mathsf{Tm}(\Gamma \rhd \sigma, \tau)}{\mathsf{lam}_{\Gamma,\sigma,\tau}(t) : \mathsf{Tm}(\Gamma, \Pi_\Gamma(\sigma,\tau))} \qquad \blacksquare$$

The strategy is the same as in the last section: we will generalise Fam(Set) to a category $\mathsf{Fam}^n(\mathsf{Set})$ consisting of telescopes of length n, and show that this category is still a Category with Families.

Definition 6.19 The category $\mathsf{Fam}^n(\mathbb{C})$ is defined by recursion over $n : \mathbb{N}$ as follows:

$$\begin{aligned} \mathsf{Fam}^0(\mathbb{C}) &:= \mathbb{C} \\ \mathsf{Fam}^{n+1}(\mathbb{C}) &:= \mathsf{Fam}\,(\mathsf{Fam}^n(\mathbb{C})) \end{aligned} \qquad \boxdot$$

Thus, $\mathsf{Fam}^1(\mathbb{C})$ is just $\mathsf{Fam}\,C$, and $\mathsf{Fam}^n(\mathbb{C})$ can be explicitly described as having objects $(n+1)$-tuples $(A, B_1, \ldots, B_n)$, where

$$
\begin{aligned}
&A : \mathsf{Set} \\
&B_1 : A \to \mathsf{Set}\ , \\
&B_2 : (x_0 : A) \to B_1(x_0) \to \mathsf{Set}\ , \\
&B_3 : (x_0 : A) \to (x_1 : B_1(x_0)) \to B_2(x_0, x_1) \to \mathsf{Set}\ , \\
&\ \vdots \\
&B_n : (x_0 : A) \to (x_1 : B_1(x_0)) \to \ldots \to B_{n-1}(x_0, \ldots, x_{n-2}) \to \mathsf{Set}\ .
\end{aligned}
$$

We see that $\mathsf{Fam}^2(\mathsf{Set})$ indeed has the right structure to model the contexts, types and terms from Example 6.18. There is a chain of projection functors

$$\begin{tikzcd}
\mathbb{C} & \mathsf{Fam}^1(\mathbb{C}) \arrow[l, "U_0"'] & \mathsf{Fam}^2(\mathbb{C}) \arrow[l, "U_1"'] & \mathsf{Fam}^3(\mathbb{C}) \arrow[l, "U_2"'] & \ldots \arrow[l] & \mathsf{Fam}^n(\mathbb{C}) \arrow[l, "U_{n-1}"']
\end{tikzcd}$$

defined by $U_k(A, B_1, \ldots, B_k, B_{k+1}) = (A, B_1, \ldots, B_k)$. Let us write $U_{i,k}$ for the composite

$$U_{i,k} := U_i \circ U_{i+1} \circ \ldots \circ U_k : \mathsf{Fam}^{k+1}(\mathbb{C}) \to \mathsf{Fam}^i(\mathbb{C})\ .$$

By iterating Proposition 4.35 ($\mathsf{Fam}\,\mathbb{C}$ has finite limits if $\mathbb{C}$ does) n times, we can prove:

Lemma 6.20 The category $\mathsf{Fam}^n(\mathsf{Set})$ has all finite limits, and all projection functors $U_{i,k}$ preserve them. □

6.2.2.1 Inductive-inductive definitions of telescopic families

We construct a Category with Families that represent telescopic inductive-inductive definitions, generalising the construction in Section 4.1.2. Before we carry out the construction for an arbitrary telescope $(A, B_1, \ldots, B_n)$, let us do the concrete case $n = 3$; the pattern should be clear from this example already. See Figure 6.2 for an overview of what is now to come.

The first piece of data needed to give an inductive-inductive definition of the telescope (A, B_1, B_2, B_3) is a functor

$$\mathsf{Arg}_{\mathsf{A}} : \mathsf{Fam}^3(\mathsf{Set}) \to \mathsf{Set}$$

which we should think of as describing the domain of the constructor for the first set A. The constructor for the family $B_1 : A \to \mathsf{Set}$ can make use of the constructor for A, and is thus represented by a functor

$$\mathsf{Arg}_{\mathsf{B}_1} : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U_{0,2}) \to \mathsf{Fam}(\mathsf{Set})$$

such that $U_0 \circ \mathsf{Arg}_{\mathsf{B}_1} = \mathsf{Arg}_{\mathsf{A}} \circ V_0$ where $V_0 : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U_{0,n}) \to \mathsf{Fam}^3(\mathsf{Set})$ is the forgetful functor $V_0(X, f) = X$, i.e. the first component of $\mathsf{Arg}_{\mathsf{B}_1}$ agrees with $\mathsf{Arg}_{\mathsf{A}}$. Just like in Section 4.1.2, we must now make sure that the dialgebra that represents the constructor for B_1, but also contains a morphism for the constructor for A, actually contains the "right" constructor for A. Once again, we do so by equalising appropriate functors: the forgetful functor $V_1 : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_1}, U_{1,2} \circ V_0) \to \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U_{0,2})$, and the functor $(V_0, U_0) : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_1}, U_{1,2} \circ V_0) \to \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{A}}, U_{0,2})$ defined by $(V_0, U_0)(X, f) = (V_0(X), U_0(f))$. This is well-defined since

$$U_0(f) : U_0(\mathsf{Arg}_{\mathsf{B}_1}(X)) \to U_0(U_{1,2}(V_0(X)))$$

but $U_0 \circ \mathsf{Arg}_{\mathsf{B}_1} = \mathsf{Arg}_{\mathsf{A}} \circ V_0$ and $U_0 \circ U_{1,2} = U_{0,2}$, hence

$$U_0(f) : \mathsf{Arg}_{\mathsf{A}}(V_0(X)) \to U_{0,2}(V_0(X))$$

and $(V_0(X), U_0(f))$ is indeed an $(\mathsf{Arg}_{\mathsf{A}}, U_{0,2})$-dialgebra. Write $\mathsf{Eq}(V_1, (V_0, U_0))$ for the equaliser category and $E_1 : \mathsf{Eq}(V_1, (V_0, U_0)) \hookrightarrow \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_1}, U_{1,2} \circ V_0)$ for the embedding. Explicitly, $\mathsf{Eq}(V_1, (V_0, U_0))$ has objects (A, B_1, B_2, B_3, c, d) where

$$\begin{aligned}&c : \mathsf{Arg}_{\mathsf{A}}(A, B_1, B_2, B_3) \to A\\&d : (x : \mathsf{Arg}_{\mathsf{A}}(A, B_1, B_2, B_3)) \to \mathsf{Arg}'_{\mathsf{B}_1}(A, B_1, B_2, B_3, c, x) \to B_1(c(x))\end{aligned}$$

where $\mathsf{Arg}'_{\mathsf{B}_1}$ is the second component of $\mathsf{Arg}_{\mathsf{B}_1}$. By passing to the equaliser category, we have made sure that the c occurring in $\mathsf{Arg}'_{\mathsf{B}_1}$ and the c occurring in B_1 are the same. Thus, this is the domain of the functor representing the constructor for B_2; we require a functor

$$\mathsf{Arg}_{\mathsf{B}_2} : \mathsf{Eq}(V_1, (V_0, U_0)) \to \mathsf{Fam}^2(\mathsf{Set})$$

such that $U_1 \circ \mathsf{Arg}_{\mathsf{B}_2} = \mathsf{Arg}_{\mathsf{B}_1} \circ V_1 \circ E_1$ – once again, the first component of $\mathsf{Arg}_{\mathsf{B}_2}$ must agree with the earlier functors. Again, we pass to the equaliser category, this time of the functor $E_1 \circ V_2$ composed of the embedding $E_1 : \mathsf{Eq}(V_1, (V_0, U_0)) \to \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_1}, U_{1,2} \circ V_0)$ and the forgetful functor $V_2 : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_2}, U_2 \circ V_0 \circ V_1 \circ E_1) \to \mathsf{Eq}(V_1, (V_0, U_0))$, and the functor $(V_1 \circ E_1, U_1)$ defined by $(V_1 \circ E_1, U_1)(X, f) = (V_1(E_1(X)), U_1(f))$. The final piece of data we require to describe our telescopic inductive-inductive definition is a functor

$$\mathsf{Arg}_{\mathsf{B}_3} : \mathsf{Eq}(E_1 \circ V_2, (V_1 \circ E_1, U_1)) \to \mathsf{Fam}^3(\mathsf{Set})$$

such that $U_2 \circ \mathsf{Arg}_{\mathsf{B}_3} = \mathsf{Arg}_{\mathsf{B}_2} \circ V_2 \circ E_2$, which describes the constructor for B_3. The category $\mathbb{E}_{\mathsf{Arg}_{\mathsf{A}}, \mathsf{Arg}_{\mathsf{B}_1}, \mathsf{Arg}_{\mathsf{B}_2}, \mathsf{Arg}_{\mathsf{B}_3}}$ whose initial object is our intended inductive-inductive definition, finally, is the equaliser of the functor $E_2 \circ V_3 : \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_3}, V_0 \circ V_1 \circ E_1 \circ V_2 \circ E_2) \to \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{B}_2}, U_2 \circ V_0 \circ V_1 \circ E_1)$ and the functor $(V_2 \circ E_2, U_2)$ defined by $(V_2 \circ E_2, U_2)(X, f) = (V_2(E_2(X)), U_2(f))$. All functors involved are summarised in a diagram in Figure 6.2.

Example 6.21 The contexts, types and terms from Example 6.18 are represented by the three functors

$$\begin{aligned}\mathsf{Arg}_{\mathsf{Ctxt}} &: \mathsf{Fam}^2(\mathsf{Set}) \to \mathsf{Set}\\\mathsf{Arg}_{\mathsf{Ty}} &: \mathsf{Dialg}(\mathsf{Arg}_{\mathsf{Ctxt}}, U_{0,1}) \to \mathsf{Fam}(\mathsf{Set})\\\mathsf{Arg}_{\mathsf{Tm}} &: \mathsf{Eq}(V_1, (V_0, U_0)) \to \mathsf{Fam}^2(\mathsf{Set})\ ,\end{aligned}$$

$$\begin{tikzcd}
\mathsf{Set} & \mathsf{Fam}^3(\mathsf{Set}) \arrow[l, "\mathsf{Arg}_{\mathrm{A}}"'] & \\
\mathsf{Fam}(\mathsf{Set}) \arrow[u, "U_0"] & \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U_{0,2}) \arrow[l, "\mathsf{Arg}_{\mathrm{B}_1}"'] \arrow[u, "V_0"'] & \\
\mathsf{Fam}^2(\mathsf{Set}) \arrow[u, "U_1"] & \mathsf{Eq}(V_1, (V_0, U_0)) \arrow[l, "\mathsf{Arg}_{\mathrm{B}_2}"'] \arrow[r, hook, "E_1"] & \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_1}, U_{1,2} \circ V_0) \arrow[ul, "V_1"'] \\
\mathsf{Fam}^3(\mathsf{Set}) \arrow[u, "U_2"] & \mathsf{Eq}(E_1 \circ V_2, (V_1 \circ E_1, U_1)) \arrow[l, "\mathsf{Arg}_{\mathrm{B}_3}"'] \arrow[r, hook, "E_2"] & \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_2}, U_2 \circ V_0 \circ V_1 \circ E_1) \arrow[ul, "V_2"'] \\
 & \mathsf{Eq}(E_2 \circ V_3, (V_2 \circ E_2, U_2)) \arrow[r, hook, "E_3"] & \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_3}, V_0 \circ V_1 \circ E_1 \circ V_2 \circ E_2) \arrow[ul, "V_3"']
\end{tikzcd}$$

Figure 6.2: Functors for telescopic inductive-inductive definitions of length $n = 3$.

defined by (we leave out the universe, as it is not covered by the current formalisation, as discussed in Example 6.18)

$$\begin{aligned}
\mathsf{Arg}_{\mathsf{Ctxt}}(\mathsf{Ctxt}, \mathsf{Ty}, \mathsf{Tm}) &= \mathbf{1} + (\Sigma\Gamma : \mathsf{Ctxt})\mathsf{Ty}(\Gamma) \\
\mathsf{Arg}_{\mathsf{Ty}}(\mathsf{Ctxt}, \mathsf{Ty}, \mathsf{Tm}, \mathsf{intro}_{\mathsf{Ctxt}}) &= (\mathsf{Arg}_{\mathsf{Ctxt}}(\mathsf{Ctxt}, \mathsf{Ty}, \mathsf{Tm}), \mathsf{Arg}'_{\mathsf{Ty}}) \\
\mathsf{Arg}_{\mathsf{Tm}}(\mathsf{Ctxt}, \mathsf{Ty}, \mathsf{Tm}, \mathsf{intro}_{\mathsf{Ctxt}}, \mathsf{intro}_{\mathsf{Ty}}) &= (\mathsf{Arg}_{\mathsf{Ty}}(\mathsf{Ctxt}, \mathsf{Ty}, \mathsf{Tm}, \mathsf{intro}_{\mathsf{Ctxt}}), \mathsf{Arg}'_{\mathsf{Tm}})
\end{aligned}$$

where (writing $\Gamma := \mathsf{intro}_{\mathsf{Ctxt}}(x)$)

$$\begin{aligned}
\mathsf{Arg}'_{\mathsf{Ty}}(x) = \mathbf{1} & & (\iota) \\
+ (\Sigma\sigma : \mathsf{Ty}(\Gamma))\mathsf{Ty}(\mathsf{intro}_{\mathsf{Ctxt}}(\mathsf{inr}(\langle\Gamma, \sigma\rangle))) & & (\Pi) \\
+ (\Sigma\Delta : \mathsf{Ctxt})(\Sigma\sigma : \mathsf{Ty}(\Delta))\mathsf{Ty}(\Gamma) \times (x \equiv \mathsf{inr}(\langle\Delta, \sigma\rangle)) & & (\mathsf{Wk})
\end{aligned}$$

and

$$\begin{aligned}
&\mathsf{Arg}'_{\mathsf{Tm}}(x, y) = \\
&\quad (\Sigma\Gamma : \mathsf{Ctxt})(\Sigma\sigma : \mathsf{Ty}(\Gamma))(x \equiv \mathsf{inr}(\langle\Gamma, \sigma\rangle) \times y \equiv \mathsf{inr}(\mathsf{inr}(\langle\Gamma, \sigma, \sigma\rangle))) & (\mathsf{top}) \\
&\quad + (\Sigma\Gamma : \mathsf{Ctxt})(\Sigma\sigma : \mathsf{Ty}(\Gamma))(\Sigma\tau : \mathsf{Ty}(\Gamma))\mathsf{Tm}(\Gamma, \tau) & (\mathsf{wk}) \\
&\qquad \times (x \equiv \mathsf{inr}(\langle\Gamma, \sigma\rangle) \times y \equiv \mathsf{inr}(\mathsf{inr}(\langle\Gamma, \sigma, \tau\rangle))) & \\
&\quad + (\Sigma\Gamma : \mathsf{Ctxt})(\Sigma\sigma : \mathsf{Ty}(\Gamma))(\Sigma\tau : \mathsf{Ty}(\mathsf{intro}_{\mathsf{Ctxt}}(\mathsf{inr}(\langle\Gamma, \sigma\rangle)))) & (\mathsf{lam}) \\
&\qquad \mathsf{Tm}(\mathsf{intro}_{\mathsf{Ctxt}}(\mathsf{inr}(\langle\Gamma, \sigma\rangle)), \tau) \times (\mathsf{intro}_{\mathsf{Ctxt}}(x) \equiv \Gamma \times y \equiv \mathsf{inr}(\mathsf{inl}(\langle\Gamma, \sigma, \tau\rangle))) & \blacksquare
\end{aligned}$$

The general construction Hopefully the pattern should be clear from the specific instance just considered, especially from Figure 6.2. Just in case, we give the general construction of an inductive-inductive definition of a telescope of length $n + 1$. Such a definition is given by a functor

$$\mathsf{Arg}_{\mathrm{A}} : \mathsf{Fam}^n(\mathsf{Set}) \to \mathsf{Set}$$

and n functors (E_i, V_i and $(V_i \circ E_i, U_i)$ will be defined below)

$$\mathsf{Arg}_{\mathrm{B}_1} : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U_{0,n-1}) \to \mathsf{Fam}(\mathsf{Set})$$
$$\mathsf{Arg}_{\mathrm{B}_{k+2}} : \mathsf{Eq}(E_k \circ V_{k+1}, (V_k \circ E_k, U_k)) \to \mathsf{Fam}^{k+2}(\mathsf{Set})$$

such that $U_k \circ \mathsf{Arg}_{\mathrm{B}_{k+1}} = \mathsf{Arg}_{\mathrm{B}_k} \circ V_k \circ E_k$ (for the purpose of this condition, let $\mathsf{Arg}_{\mathrm{B}_0} := \mathsf{Arg}_{\mathrm{A}}$). Let us write $\mathsf{dom}(\mathsf{Arg}_{\mathrm{B}_k})$ for the domain of the functor $\mathsf{Arg}_{\mathrm{B}_k}$. We now introduce the rest if the functors occurring in the specification of $\mathsf{Arg}_{\mathrm{B}_k}$:

- The functor V_0 is the forgetful functor $V_0 : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U_{0,n}) \to \mathsf{Fam}^n(\mathsf{Set})$ defined by $V_0(X, f) = X$, and V_{k+1} is the forgetful functor

  $$V_{k+1} : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_{k+1}}, U_{k+1,n-1} \circ V_0 \circ E_0 \circ \ldots V_k \circ E_k) \to \mathsf{dom}(\mathsf{Arg}_{\mathrm{B}_{k+1}})$$

  once again defined by $V_{k+1}(X, f) = X$.

- Let $E_0 := \mathsf{Id} : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U_{0,n-1}) \to \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{A}}, U_{0,n-1})$, and let

  $$E_{k+1} : \mathsf{Eq}(E_k \circ V_{k+1}, (V_K \circ E_k, U_k)) \hookrightarrow \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_{k+1}}, U_{k+1,n-1} \circ V_{0,k})$$

  (where we have written $V_{0,k} := V_0 \circ E_0 \circ \ldots V_k \circ E_k$) be the embedding given by the equaliser.

- Finally, the functor

  $$(V_K \circ E_k, U_k) : \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_{k+1}}, U_{k+1,n-1} \circ V_{0,k}) \to \mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_k}, U_{k,n-1} \circ V_{0,k-1})$$

  is defined by $(V_K \circ E_k, U_k)(X, f) = (V_K(E_k(X)), U_k(f))$. This is well-defined, since

  $$U_k(f) : U_k(\mathsf{Arg}_{\mathrm{B}_{k+1}}(X)) \to U_k(U_{k+1,n-1}(V_{0,k}(X)))$$

  but $U_k \circ \mathsf{Arg}_{\mathrm{B}_{k+1}} = \mathsf{Arg}_{\mathrm{B}_k} \circ V_k \circ E_k$, hence

  $$U_k(f) : \mathsf{Arg}_{\mathrm{B}_k}(V_k(E_k(X))) \to U_{k,n-1}(V_{0,k-1}(V_k(E_k(X))))$$

  and $(V_K(E_k(X)), U_k(f))$ is an object in $\mathsf{Dialg}(\mathsf{Arg}_{\mathrm{B}_k}, U_{k,n-1} \circ V_{0,k-1})$.

The reader so inclined can check that none of these definitions are circular – e.g. E_k, V_k are only defined in terms of E_i and V_i for $i < k$.

Given such a sequence of functors $\mathsf{Arg_A}$, $\mathsf{Arg_{B_1}}$, …, $\mathsf{Arg_{B_n}}$, we define the category $\mathbb{E}_{\mathsf{Arg_A},\vec{\mathsf{Arg_{B_i}}}}$ to be the equaliser category $\mathbb{E}_{\mathsf{Arg_A},\vec{\mathsf{Arg_{B_i}}}} := \mathsf{Eq}(E_{n-1} \circ V_n, (V_{n-1} \circ E_{n-1}, U_{n-1}))$. It has as objects $n+1$-tuples $(A, B_1, \ldots, B_n)$ where

$$\begin{aligned}
&A : \mathsf{Set}\\
&B_1 : A \to \mathsf{Set}\\
&B_2 : (x : A) \to B_1(x) \to \mathsf{Set}\\
&\qquad\qquad\vdots
\end{aligned}$$

together with appropriate "constructors"

$$\begin{aligned}
\mathsf{intro_A} &: \mathsf{Arg_A}(A, B_1, \ldots, B_n) \to A\\
\mathsf{intro}_{B_1} &: (x : \mathsf{Arg_A}(A, B_1, \ldots, B_n)) \to \mathsf{Arg_{B_1}}(A, B_1, \ldots, B_n, \mathsf{intro_A}, x) \to B_1(\mathsf{intro_A}(x))\\
&\ \ \vdots
\end{aligned}$$

By applying first Lemma 6.20, and then Theorem 4.37 and a variant of Corollary 4.39 repeatedly, we know that $\mathbb{E}_{\mathsf{Arg_A},\vec{\mathsf{Arg_{B_i}}}}$ is a category with families, and that Theorems 4.41 and 4.42 apply. The initial object in $\mathbb{E}_{\mathsf{Arg_A},\vec{\mathsf{Arg_{B_i}}}}$ (if it exists, this of course depends on the functors $\mathsf{Arg_A}$, $\mathsf{Arg_{B_i}}$ chosen being suitably strictly positive etc) is thus the data type we want.

Remark 6.22 Dybjer and Setzer [2006] extended the theory of inductive-recursive definitions to *indexed* inductive-recursive definitions, where a family $U : I \to \mathsf{Set}$ of universes together with decoding functions $T : (i : I) \to U(i) \to D$ are defined all at once for some fixed index set $I : \mathsf{Set}$. With this extension, we have in particular achieved the same thing for inductive-inductive definitions: just let the first set A be an isomorphic copy of the index set I in question, i.e. be given by a constructor

$$\mathsf{intro_A} : I \to A \ .$$

The inverse $I \to A$ can be defined via the elimination principle for A, and (after applying the isomorphism) we now have

$$\begin{aligned}
&B : I \to \mathsf{Set}\\
&C : (i : I) \to B(i) \to \mathsf{Set}
\end{aligned}$$

i.e. an indexed inductive-inductive definition.

6.3 Summary

We have explored two different extensions to the theory of inductive-inductive definitions. The first combines inductive-inductive and inductive-recursive definitions into one theory. This is closer to what is actually possible in Agda today. By adapting Dybjer and Setzer's set-theoretical model for inductive-recursive definitions, we could show that also the combined theory is consistent.

Secondly, we extended the theory to cope with the examples we have already informally introduced. We first generalised the way B can be indexed over A from $B : A \to \mathsf{Set}$ to $B : F(A) \to \mathsf{Set}$ for any pullback-preserving functor $F : \mathsf{Set} \to \mathsf{Set}$ (with, of course, $F = \mathsf{Id}$ being a special case). In an orthogonal direction, we then generalised the number of simultaneously defined sets and families of sets from two to arbitrarily many. Both these generalisations made essential use of the categorical framework developed in Chapter 4.

CHAPTER 7

Case studies

Contents

In this chapter, we carry out two larger developments making essential use of inductive-inductive definitions. We hope to demonstrate that inductive-inductive definitions play a natural rôle in everyday mathematical practise. The first case study investigates Conway's surreal numbers in type theory. The second case study develops a theory of positive inductive-recursive definitions, invented by Lorenzo Malatesta and Neil Ghani.

The work on positive inductive-recursive definitions has been published in the proceedings of CALCO 2013 [Ghani, Malatesta, and Nordvall Forsberg, 2013a]. Here we concentrate on the use of inductive-inductive definitions to develop the theory of positive induction-recursion. See Malatesta's thesis [2013] for a treatment of the theory as a whole.

7.1 Conway's surreal numbers

In this section, we give a larger example of how inductive-inductive definitions can be naturally used to develop ordinary mathematics in type theory.

7.1.1 Introduction

The surreal numbers [Conway, 2001; Knuth, 1974] classically form a totally ordered Field[1] which contains the real numbers as an ordered subfield and the ordinals as an ordered substructure. The usual route of constructing the real numbers goes $\mathbb{N} \rightsquigarrow \mathbb{Z} \rightsquigarrow \mathbb{Q} \rightsquigarrow \mathbb{R}$, with four different sets of arithmetical operations and order relations that must be proven to coincide, and the natural numbers are "reconstructed" three times

[1] i.e. a field whose domain is a proper class.

as certain integers, rationals and real numbers. Conway, on the other hand, constructs all of the surreal numbers – which include all of the number classes mentioned above, and more – with just one definition!

Conway is working in set theory with unrestricted use of the law of excluded middle and the Axiom of Choice. We will instead explore the surreal numbers from a constructive and type-theoretic point of view, and we will see that they together with their order relation naturally form an inductive-inductive definition. We are not the first to do so: Rosemeier [2001] investigates the surreal numbers from a constructive point of view, working in Bishop's tradition in informal set theory, and Mamane [2006] develops the theory of surreal numbers in the proof assistant Coq (with postulated classical axioms). Both Rosemeier and Mamane follow Conway and work with an encoding of surreal numbers along the lines of Section 5.3, whereas we will make use of inductive-inductive definitions and consider a more faithful representation. The Homotopy Type Theory book [The Univalent Foundations Program, 2013] gives a *higher* inductive-inductive definition of the surreal numbers, taking inspiration from our example in Nordvall Forsberg and Setzer [2012].

We also take the opportunity to show how inductive-inductive definitions can be used in real world formal developments in the proof assistant and functional programming language Agda (see Section 2.2). Agda is not based on any formal theory of data types – in particular, there is no formal proof that Agda implements a consistent theory – but we will see that Agda actually works quite well for this purpose. In fact, this entire section is a literate Agda file[2]. We declare the start of the module:

```
module surreal where
```

and we include some standard definitions from the standard library, making sure to hide definitions whose names will clash:

```
open import Data.Product   hiding (∃)                   -- sigma types
open import Data.Empty                                  -- empty type
open import Data.Unit      hiding (_≤_)                 -- unit type
open import Data.Bool      hiding (T)                   -- Booleans
open import Data.Nat       hiding (_<_; _>_; _≤_; _≥_)  -- natural numbers
open import Data.Integer   hiding (_≤_; -_)             -- integers
open import Data.Sum                                    -- disjoint sum
```

7.1.2 Surreal numbers, informally

Before we dive into the type theory, let us take a step back and appreciate Conway's set-theoretical definition. As a starting point, let us consider the definition of a Dedekind cut, used as one possible way to construct $\mathbb{R}$ from $\mathbb{Q}$:

[2] The code in this section type checks with Agda 2.3.2.2 and version 0.7 of the standard library.

Definition 7.1 A *Dedekind cut* (L, R) consists of two non-empty sets $L, R \subseteq \mathbb{Q}$ such that $L \cup R = \mathbb{Q}$, all elements of L are less than all elements of R and L contains no greatest element. ■

We now change the definition so that it makes sense without having constructed the rational numbers $\mathbb{Q}$ first. The resulting defined object will be called a *surreal number*.

- According to our policy of avoiding $\mathbb{Q}$, L and R should not be sets of rational numbers. Instead, we make them sets of already constructed surreal numbers.

- If L and R are not subsets of $\mathbb{Q}$, it makes no sense to demand that $L \cup R = \mathbb{Q}$, hence we remove that condition.

- Now that L and R are sets of already constructed numbers, we better have a way to start the construction, which is hard if we demand that all sets are non-empty, or L infinite. Hence we remove these two conditions as well.

We end up with the following definition of the surreal numbers, where we have changed the notation from (L, R) to $\{L \mid R\}$ (following Conway [2001]) to avoid confusing surreal numbers and Dedekind cuts:

Definition 7.2 A *surreal number* $\{L \mid R\}$ consists of two sets L, R of surreal numbers such that all elements of L are less than all elements of R. The class of all surreal numbers is denoted No. □

For a surreal number X, we write $X = \{X_L \mid X_R\}$, and we will write x^L for a typical element of X_L (called a *left option* of X) and x^R for a typical element of X_R (called a *right option*). With this notation, the condition on a surreal number $\{L \mid R\}$ can be written

$$(\forall x^L \in L)(\forall x^R \in R)\, x^L < x^R \ . \tag{7.1}$$

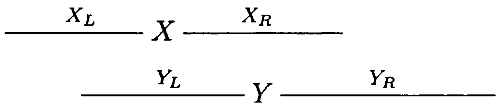
The intuition is that $X = \{X_L \mid X_R\}$ is the "simplest" number above all the numbers in X_L and below all the numbers in X_R:

$$\overset{X_L}{\rule{3cm}{0.4pt}}\; X \;\overset{X_R}{\rule{2cm}{0.4pt}}$$

This explains condition (7.1): if X is to lie in between X_L and X_R, there cannot be an element $x^L \in X_L$ which is larger or equal to an element $x^R \in X_R$, or we would have $x^R \le x^L < X < x^R$, i.e. $x^R < x^R$, which should be a contradiction.

The attentive reader might have noticed that something is not right: equation (7.1) made sense for Dedekind cuts, because in that situation, the elements x^L and x^R were rational numbers for which we know what the order relation $<$ is. However, since we just defined the surreal numbers, we also need to define what it means for one surreal number to be smaller than another, and this has to be done simultaneously with the original definition, since it refers to $<$; this is an inductive-inductive definition!

Conway prefers to define $\le$ instead of $<$. In a classical setting, this is harmless, as we expect $x < y \Leftrightarrow \neg(y \le x)$, but we will get to this point in Section 7.1.4. The intuition once again comes from seeing X as the simplest number between X_L and X_R: if $X \leq Y$, then it cannot be the case that X is greater than or equal to any $y^R \in Y_R$, and it also cannot be the case that any $x^L \in X_L$ is greater than or equal to Y:



Formally, Conway defines:

Definition 7.3 Let $X = \{X_L \mid X_R\}$ and $Y = \{Y_L \mid Y_R\}$ be surreal numbers. We have $X \leq Y$ if

$$(\forall y^R \in Y_R)\neg(y^R \leq X) \text{ and } (\forall x^L \in X_L)\neg(Y \leq x^L) \ . \qquad \blacksquare$$

7.1.2.1 Examples of surreal numbers

We present some examples of surreal numbers in set-theoretical notation. We will write $\{\ldots, a, b, c, \ldots \mid \ldots, d, e, f, \ldots\}$ instead of $\{\{\ldots, a, b, c, \ldots\} \mid \{\ldots, d, e, f, \ldots\}\}$.

Examples 7.4

(i) The simplest surreal number of them all, and the one needed to get off the ground, is $0_{\mathsf{No}} := \{\varnothing \mid \varnothing\}$. It is trivially a surreal number, since it is certainly true that

$$(\forall x^L \in \varnothing)(\forall x^R \in \varnothing)\, x^L < x^R \ .$$

(ii) Having constructed 0_{No}, we can now construct $1_{\mathsf{No}} := \{0_{\mathsf{No}} \mid \varnothing\}$ and $-1_{\mathsf{No}} := \{\varnothing \mid 0_{\mathsf{No}}\}$, both again trivially satisfying (7.1).

(iii) In general, we can define $(-)_{\mathsf{No}} : \mathbb{Z} \to \mathsf{No}$ by

$$\begin{aligned} 0_{\mathsf{No}} &:= \{\varnothing \mid \varnothing\} & \\ (n+1)_{\mathsf{No}} &:= \{n_{\mathsf{No}} \mid \varnothing\} & n \geq 0 \\ (n-1)_{\mathsf{No}} &:= \{\varnothing \mid n_{\mathsf{No}}\} & n \leq 0 \end{aligned}$$

(iv) We do not have to stop at finite numbers: we can define $\omega := \{0_{\mathsf{No}}, 1_{\mathsf{No}}, 2_{\mathsf{No}}, \ldots \mid \varnothing\}$. In general, ordinals can be identified with surreal numbers of the form $\{L \mid \varnothing\}$.

(v) Real numbers can be identified with surreal numbers x such that $-n < x < n$ for some integer n (i.e. x is finite), and $x \equiv \{x-1, x-\frac{1}{2}, x-\frac{1}{3}, \ldots \mid x+1, x+\frac{1}{2}, x+\frac{1}{3}, \ldots\}$ (i.e. x is equivalent to a "rational cut") where we have written $X \equiv Y$ for $X \leq Y \wedge Y \leq X$ (we will return to this relation in Section 7.1.6). ■

7.1.3 Set theory in Type Theory

In order to define surreal numbers in Type Theory similar to the way they are defined in Definitions 7.2 and 7.3, we need to be able to talk about subsets of surreal numbers. We will follow Azcel's interpretation of CZF in Type Theory [Aczel, 1978], and represent a subset X of the set A as an index set I_X and a function $f_X : I_X \to A$, which we think of as picking out the elements of X. In other words, we model subsets of A as objects of Fam A. Since we want to keep things small, we cannot use arbitrary sets as index sets. Instead, we localise the definition to a universe

```
data U : Set
T : U → Set
```

in the style of Hancock [Ghani and Hancock, 2012]. The exact details of the universe U can be left open, but it should at least contain **0**, **1**, **2** and the natural numbers, and be closed under disjoint unions – we also close it under dependent pairs and functions:

```
data U where
  ℕ' ⊥' ⊤' Bool' : U
  Π' Σ' : (a : U) → (b : T a → U) → U
  _⊎'_ : U → U → U

T ℕ' = ℕ
T ⊥' = ⊥
T ⊤' = ⊤
T Bool' = Bool
T (Π' a b) = (x : T a) → T (b x)
T (Σ' a b) = Σ (T a) (λ x → T (b x))
T (a ⊎' b) =  T a ⊎ T b
```

A subset X of A is represented by an (U, T)-localised family of As, i.e. a U-small index set X ind : U and an element function X el : X ind → A:

```
record 𝒫 (A : Set) : Set where
  constructor subset
  field
    _ind : U
    _el_ : T _ind → A

  infix 6 _el_
open 𝒫
```

For example, we can define the empty subset of any set by

```
∅ : {A : Set} → 𝒫 A
∅ = subset ⊥' ⊥-elim
```

and singletons and disjoint unions of subsets by

```
⦃_⦄ : {A : Set} → A → 𝒫 A
⦃ x ⦄ = subset ⊤' (λ _ → x)

_⊎_ : {A : Set} → 𝒫 A → 𝒫 A → 𝒫 A
(subset I Xi) ⊎ (subset J Yi) = subset (I ⊎' J) [ Xi , Yi ]
```

Remark 7.5 Unfortunately, Agda has already reserved the symbols { and } for implicit arguments. We will squint and use the similar looking ⦃ and ⦄ instead, both for the singleton subset above, and for the surreal number $\{L \mid R\}$ later.

We also define quantifiers for elements of a subset X of A. We say that $\forall[x \in X]\varphi$ holds for a property $\varphi : A \to \mathsf{Set}$ if $\varphi(X \text{ el } i)$ holds for all $i : \mathsf{T}(X \text{ ind})$, and similarly for $\exists[x \in X]\varphi$.

```
∀' : {A : Set} → (X : 𝒫 A) → (φ : A → Set) → Set
∀' {A} X φ = (x : T (X ind)) → φ (X el x)

record ∃ {A : Set} (X : 𝒫 A) (φ : A → Set) : Set where
  constructor exists
  field
    witness : T (X ind)
    proof : ϕ (X el witness)
```

By using Agda's syntax facility (which should be read from right to left), we can get a nice notation for the quantifiers:

```
syntax ∀' X (λ x → ϕ) = ∀[ x ∈ X ] ϕ
syntax ∃ X (λ x → ϕ) = ∃[ x ∈ X ] ϕ
```

7.1.4 Surreal numbers as an inductive-inductive definition

We now translate Conway's definition of the surreal numbers into Type Theory. We already observed that this will need to be an inductive-inductive definition and not a simple inductive definition, since the set of surreal numbers No and the order relation ≤: No → No → Set are mutually defined.

However, there is a second complication: we would like to define also ≤ inductively, but looking at Definition 7.3 again, we see that ≤ appears negatively (literally!) in its defining formula

$$(\forall y^R \in Y_R)\neg(y^R \leq X) \text{ and } (\forall x^L \in X_L)\neg(Y \leq x^L) \ .$$

Following Rosemeier [2001] (and also Mamane [2006]), we instead simultaneously define ≥ and its negation, which we write <. This makes sense from a constructive point of view anyway, as we would like to define < using positive information (compare with the notion of *apartness relation* in constructive real analysis [Bishop and Bridges, 1985]). Thus, we simultaneously define

```
data No : Set
data _≥_ : No → No → Set
data _<_ : No → No → Set
```

using inductive-inductive definitions in the following way:

```
data No where
  ⦃_|_⦄_ : (XL : 𝒫 No) → (XR : 𝒫 No) →
              ∀[ xl ∈ XL ] (∀[ xr ∈ XR ] (xl < xr))
                → No

data _≥_ where
  geq :  {XL : 𝒫 No} → {XR : 𝒫 No} →
         {p : ∀[ xl ∈ XL ] (∀[ xr ∈ XR ] (xl < xr))} →
         {YL : 𝒫 No} → {YR : 𝒫 No} →
         {q : ∀[ yl ∈ YL ] (∀[ yr ∈ YR ] (yl < yr))} →
         (let      X = ⦃ XL | XR ⦄ p
                   Y = ⦃ YL | YR ⦄ q
         in
             (∀[ xr ∈ XR ] Y < xr) →
             (∀[ yl ∈ YL ] yl < X)   →
         X ≥ Y)

data _<_ where
  ltr :  {XL : 𝒫 No} → {XR : 𝒫 No} →
         {p : ∀[ xl ∈ XL ] (∀[ xr ∈ XR ] (xl < xr))} →
         {Y : No} →
         (let X = ⦃ XL | XR ⦄ p
            in
              ∃[ xr ∈ XR ] Y ≥ xr →
            X <  Y)
  ltl :  {X : No}
         {YL : 𝒫 No} → {YR : 𝒫 No} →
         {q : ∀[ yl ∈ YL ] (∀[ yr ∈ YR ] (yl < yr))} →
```

```
        (let Y = ⟨ YL | YR ⟩ q
          in
            ∃[ yl ∈ YL ] yl ≥ X →
          X < Y)
```

Notice that $X<Y$ is just the positive version of $\neg(Y \geq X)$. Indeed, if we have positive information that $X<Y$, then we know that it cannot be the case that $X \geq Y$ – the negative statement of $X<Y$.

Lemma 7.6 If $X<Y$, then $\neg(X \geq Y)$.

Proof. We prove the statement by induction on $X<Y$ and $X \geq Y$. Assume $X \geq Y$, i.e. $Y<x^R$ for all $x^R \in X_R$ and $y^L<X$ for all $y^L \in Y_L$. Also assume $X<Y$, which can happen for two possible reasons:

If $X<Y$ because $Y \geq x^R$ for some $x^R \in X_R$, then both $Y<x^R$ and $Y \geq x^R$ for that $x^R \in X_R$, which is a contradiction by the induction hypothesis. Similarly, if $X<Y$ because $y^L \geq X$ for some $y^L \in Y_L$, then this and $y^L<X$ once again leads to a contradiction by the induction hypothesis.

The Agda proof of the lemma is short and sweet:

```
<-to-≥→⊥ : {X Y : No} → X < Y → X ≥ Y → ⊥
<-to-≥→⊥ (ltr (exists xri y≥xr)) (geq y<xr yl<x) = <-to-≥→⊥ (y<xr xri) y≥xr
<-to-≥→⊥ (ltl (exists yli yl≥x)) (geq y<xr yl<x) = <-to-≥→⊥ (yl<x yli) yl≥x
```

□

In the following, we will often only give Agda proofs, as they are quite readable on their own. We do not expect the converse of the lemma to hold without classical logic, of course. For convenience, we define

```
_≤_ : No → No → Set
x ≤ y = y ≥ x

_>_ : No → No → Set
x > y = y < x

infix 4 _<_ _>_
infix 5 _≥_ _≤_
```

as well as projection functions

```
_L : No → 𝒫 No
(⟨ XL | _ ⟩ _) L = XL

_R : No → 𝒫 No
```

```
(❪ _ | XR ❫ _) R = XR
```

7.1.4.1 Examples of surreal numbers in Type Theory

Let us take a look at the type theoretical versions of the examples of surreal numbers from Section 7.1.2.1.

Examples 7.7

(i) The surreal number $0_{\mathsf{No}} = \{\varnothing \mid \varnothing\}$ becomes

```
zeroS : No
zeroS = ❪ ∅ | ∅ ❫ (λ x e → ⊥-elim e)
```

in the type-theoretical setting, where the empty subset ∅ := subset ⊥' ⊥-elim was defined in Section 7.1.3. Notice how the trivial proof that

$$(\forall x^L \in \varnothing)(\forall x^R \in \varnothing)\, x^L < x^R$$

is just an application of ex falso quod libet.

(ii) In the same straightforward way, we can define

```
oneS : No
oneS = ❪ ❪ zeroS ❫ | ∅ ❫ (λ x e → ⊥-elim e)

-oneS : No
-oneS = ❪ ∅ | ❪ zeroS ❫ ❫ (λ e x → ⊥-elim e)
```

(iii) We can also represent the integers as type-theoretic surreal numbers. This is most conveniently presented as an embedding of $\mathbb{Z}$ into No, defined by recursion over the integer:

```
ιℤ : ℤ → No
ιℤ -[1+ zero ] = -oneS
ιℤ -[1+ ℕ.suc n ] = ❪ ∅ | ❪ ιℤ -[1+ n ] ❫ ❫ (λ e x → ⊥-elim e)
ιℤ (+ zero) = zeroS
ιℤ (+ ℕ.suc n) = ❪ ❪ ιℤ (+ n ) ❫ | ∅ ❫ (λ x e → ⊥-elim e)
```

(iv) As a special case of the embedding $\iota_{\mathbb{Z}}$ of the integers, we get an embedding of the natural numbers

```
ιℕ : ℕ → No
ιℕ n = ιℤ (+ n)
```

which we can use to define the first infinite ordinal

```
ω : No
ω = ⦇ subset ℕ' ιℕ | ∅ ⦈ (λ x e → ⊥-elim e)
```

In general, remember that Conway defined the ordinals to be surreal numbers of the form $\{L \mid \varnothing\}$. We can define the corresponding predicate

```
is-ordinal : No → Set
is-ordinal X = ∀[ x ∈ X R ] ⊥
```

and indeed, the identity function is a proof that ω is an ordinal. This is is most likely not a constructively useful notion of an ordinal.

(v) The corresponding predicate for being a real number can also be defined, but requires some more infrastructure dealing with rational numbers and addition of surreal numbers. We omit it here. ■

7.1.5 Properties and operations

Let us now demonstrate that our definition of the surreal numbers is useful by proving some (expected) properties about them and defining arithmetical operations on them; both activities will have a distinct inductive-inductive flavour. To do so, we will of course use the elimination rules, but this time in the form of Agda's built in support for dependent pattern matching [Coquand, 1992].

In short, Agda allows us to define a function f from an inductively defined data type by giving its value on the constructors of the data type. Furthermore, f may be used for recursive calls on structurally smaller arguments. If f is a function of multiple arguments, it is not necessary for all arguments to decrease [Abel and Altenkirch, 2002], which will often be useful for our purposes. Conway [2001, p. 5] writes

> In general when we wish to establish a proposition $P(x)$ for all numbers x, we will prove it inductively by deducing $P(x)$ from the truth of all the propositions $P(x^L)$ and $P(x^R)$. [...] When proving propositions $P(x,y)$ involving two variables, we may use *double induction*, deducing $P(x,y)$ from the truth of all propositions of the form $P(x^L,y)$, $P(x^R,y)$, $P(x,y^L)$, $P(x,y^R)$ (and, if necessary, $P(x^L,y^L)$, $P(x^L,y^R)$, $P(x^L,y^R)$, $P(x^R,y^L)$). Such multiple inductions can be justified in the usual way in terms of repeated single inductions.

The point is that these kind of inductive arguments, including double induction, is exactly what Agda's pattern matching gives us. Goguen et al. [2006] show how dependent pattern matching can be translated into standard eliminators (plus 'uniqueness of identity proofs') for inductive families; in particular, this includes the justification in terms of repeated single inductions that Conway alludes to.

As a warm-up, and sanity check that our definition is not completely wrong, let us prove Conway's [2001] Theorem 0, which says that the order of the surreal numbers behave as we intuitively described it in Section 7.1.2:

Lemma 7.8 (Conway's Theorem 0) For all surreal numbers $X = \{X_L \mid X_R\}$ we have

(i) $X \geq X$,

(ii) $X < x^R$ for all $x^R \in X_R$,

(iii) $x^L < X$ for all $x^L \in X_L$, and

(iv) not $X < X$.

Proof. We simply prove the lemma by giving Agda definitions of the right type:

```
refl-≥ : {X : No} → X ≥ X
refl-≥ {⟨ XL | XR ⟩ p} = geq (λ xri → ltr (exists xri refl-≥))
   (λ xli → ltl (exists xli refl-≥))

Thm0ii : {X : No} → ∀[ xr ∈ X R ] (X < xr)
Thm0ii {⟨ XL | XR ⟩ p} xri = ltr (exists xri refl-≥)

Thm0iii : {X : No} → ∀[ xl ∈ X L ] (xl < X)
Thm0iii {⟨ XL | XR ⟩ p} xli = ltl (exists xli refl-≥)

irrefl-< : {X : No} → (X < X) → ⊥
irrefl-< (ltr (exists xri x>xr)) = <-to-≥→⊥ (Thm0ii xri) x>xr
irrefl-< (ltl (exists yli yl>y)) = <-to-≥→⊥ (Thm0iii yli) yl>y
```

In the proof irrefl-< of (iv), we used <-to-≥→⊥, the proof of Lemma 7.6. □

Let us now see that we can also define operations on the surreal numbers, such as e.g. negation. Conway usually defines the operation first, and then later verifies that e.g. if X is a surreal number – i.e. satisfies the order condition (7.1) – then so is $-X$. Since we have baked in the order condition in our inductive-inductive definition of the surreal numbers, we cannot allow ourselves to do the same. Instead, we must prove that the order condition is satisfied simultaneously as we the define the operation. Thankfully, this is exactly what the (general) elimination rule for inductive-inductive definitions allow us to do.

Proposition 7.9 Conway's construction of the negation of a surreal number

$$-\{X_L \mid X_R\} := \{-X_L \mid -X_R\}$$

where $-X_i = \{-x^i : x^i \in X_i\}$ can be carried out in Type Theory, and we have

$$X \leq Y \text{ implies } -Y \leq -X$$
$$X < Y \text{ implies } -Y < -X$$

Proof. We define

```
- : No → No
lemma–≤ : {X Y : No} → X ≤ Y → - Y ≤ - X
lemma–< : {X Y : No} → X < Y → - Y < - X
```

simultaneously – note how - appears already in the type of lemma–≤ and lemma–<. The defining equations are

```
- ((⦇ XL | XR ⦈ p) = let
   -XL = subset (XL ind) (λ xli → - (XL el xli))
   -XR = subset (XR ind) (λ xri → - (XR el xri))
   in
   ⦇ -XR | -XL ⦈ (λ xri xli → lemma–< (p xli xri))

lemma–≤ (geq y¬>xr yl¬>x) = geq (λ yli → lemma–< (yl¬>x yli))
      (λ xri → lemma–< (y¬>xr xri))

lemma–< (ltr (exists xri xr<y)) = ltl (exists xri (lemma–≤ xr<y))
lemma–< (ltl (exists yli x<yl)) = ltr (exists yli (lemma–≤ x<yl))
```

□

We see that indeed $-(\iota_{\mathbb{Z}} x) = \iota_{\mathbb{Z}}(-_{\mathbb{Z}} x)$, at least morally – the formal statement requires function extensionality because of our higher-order representation of subsets.

We have already seen that ≤ is reflexive in Lemma 7.8. Let us now prove that it is also transitive. Also Conway proved this, of course, but using classical logic. Since we are working constructively, we have to do a little more work.

Lemma 7.10 (Conway's Theorem 1) ≤ is transitive, i.e. if $X \leq Y$ and $Y \leq Z$ then $X \leq Z$.

Proof. In order to be able to handle recursive calls, we define

```
trans-≤     : {X Y Z : No} → X ≤ Y → Y ≤ Z → X ≤ Z
trans-≤-<   : {X Y Z : No} → X ≤ Y → Y < Z → X < Z
trans-<-≤   : {X Y Z : No} → X < Y → Y ≤ Z → X < Z
```

simultaneously by

```
trans-≤ (geq X<yr xl<Y) (geq Y<zr yl<Z)
  = geq (λ zri → trans-≤-< (geq X<yr xl<Y) (Y<zr zri))
  (λ xli → trans-<-≤ (xl<Y xli) (geq Y<zr yl<Z))

trans-≤-< (geq X<yr xl<Y) (ltr (exists yri Z≥yr)) = trans-<-≤ (X<yr yri) Z≥yr
trans-≤-< p (ltl (exists zli zl≥Y)) = ltl (exists zli (trans-≤ p zl≥Y))

trans-<-≤ (ltr (exists xri Y≥xr)) q = ltr (exists xri (trans-≤ Y≥xr q))
trans-<-≤ (ltl (exists yli yl≥X)) (geq Y<zr yl<Z) = trans-≤-< yl≥X (yl<Z yli)
```

□

We could go on and define addition, multiplication and division, but hopefully, we have already been given a taste of what working with inductive-inductive definitions in Agda is like. We are also fast approaching the limits of what Agda and its termination checker can handle. It is nevertheless quite pleasing that so much is possible to do in Agda even today, without the developers giving any special care to inductive-inductive definitions.

7.1.6 Discussion

Numbers and games Conway [2001] first defines the surreal numbers as we have indicated above. It is then anticipated that numbers being defined simultaneously with their ordering relation might make certain people uncomfortable, and the "formal" development of surreal numbers is divided into three stages:

(i) *Games* are defined as numbers without an order condition, i.e. a game is given by two sets of games. (Games are called games, as they are used in the second half of the book to analyse strategies for mathematical games such as Nim).

(ii) The order relation is then defined on games, using the same formula as before.

(iii) Finally surreal numbers are defined to be those games that satisfy the order condition.

Mamane [2006] follows the same route, since inductive-inductive definitions are not available in Coq. There are interesting parallels with the translation of inductive-inductive definitions into indexed inductive definitions in Section 5.3, but here the "prenumbers" (i.e. games) are of a particularly simple form where there is no mutual dependency at all. It should be clear that there is no need to jump through such hoops to justify the theory – inductive-inductive definitions are justified in their own right.

Proving properties of surreal numbers The attentive reader has noticed that we have not proven many properties about surreal numbers not involving the order relation on them. This stems from the fact that we have represented subsets as functions. To prove equalities between surreal numbers, we need to prove equalities between functions, and for that we usually need function extensionality. Even a simple statement such as $-(-X) = X$, which has the following informal one-liner proof

$$-(-\{x^L \mid x^R\}) = -\{-x^R \mid -x^L\} = \{-(-x^L) \mid -(-x^R)\} = \{x^L \mid x^R\}$$

fails because we need function extensionality to apply the induction hypothesis in the last step. Thus, Agda (or our representation of subsets) is not quite adequate for a complete treatment of the surreal numbers.

Equality of surreal numbers We proved that $\leq$ is reflexive and transitive, so a natural question is if the relation is also anti-symmetric (i.e. $X{\leq}Y$ and $Y{\leq}X$ implies $X = Y$), as that would make $\leq$ into a partial order. The answer is both no and yes. No, because it is not true, and yes, because Conway declares that two surreal numbers X and Y are actually equal if $X{\leq}Y$ and $Y{\leq}X$, thus forcing $\leq$ to be a partial order (in fact, the order is total if and only if the law of excluded middle holds [Rosemeier, 2001, Prop. 1.9]). This is also needed to validate certain arithmetical laws; for instance, the equation $X + (-X) = 0$ does not hold up to propositional equality, but only up to the equivalence relation mentioned above.

The traditional type theoretical solution is thus to form a setoid of surreal numbers, with equivalence relation $X \sim Y$ iff $X{\leq}Y$ and $Y{\leq}X$. By switching to a setoid of surreal numbers, we also get function extensionality, but it remains to be seen how much extra book keeping is needed.

The Homotopy Type Theory book [The Univalent Foundations Program, 2013] instead advocates the use of a *higher* inductive-inductive definition, where the set of surreal numbers and the order relation is constructed simultaneously with new (non-canonical) constructors for the identity type, which forces the relation $\sim$ above to be logically equivalent to equality. At the time of writing, there is no computational interpretation of such higher inductive definitions.

7.2 Positive inductive-recursive definitions

As yet another example of how inductive-inductive definitions are useful, we return to Dybjer and Setzer's theory IR of inductive-recursive definitions. In Section 3.2.2, the syntax of inductive-recursive definitions was presented as an inductive definition. The definitions were then given semantics as initial algebras of endofunctors on Fam $|\mathbb{C}|$, i.e. endofunctors on the category of families of objects from some discrete category $|\mathbb{C}|$. We will see that if we upgrade the inductive definition of the syntax to an inductive-inductive one, then the discreteness condition can be lifted, i.e. we get a theory of data types, which we call *positive inductive-recursive definitions*, whose semantics are given as initial algebras of endofunctors on Fam $\mathbb{C}$ for an arbitrary category $\mathbb{C}$. We recover ordinary inductive-recursive definitions as the special case when $\mathbb{C}$ is discrete. We also extend Dybjer and Setzer's model construction to our setting.

7.2.1 The semantics of IR, revisited

In Section 3.2.2, we recalled Dybjer and Setzer's system of codes for inductive-recursive definitions. We then gave a semantics by defining two functions

$$\begin{aligned}\mathsf{Arg}_{\mathsf{IR}}(\gamma) &: (U : \mathsf{Set}) \to (T : U \to D) \to \mathsf{Set}\\ \mathsf{Fun}_{\mathsf{IR}}(\gamma) &: (U : \mathsf{Set}) \to (T : U \to D) \to \mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T) \to D\end{aligned}$$

for each code $\gamma : \mathsf{IR}(D)$. We also remarked that, using extensional type theory, $\mathsf{Arg}_{\mathsf{IR}}(\gamma)$ and $\mathsf{Fun}_{\mathsf{IR}}(\gamma)$ can be combined and extended to a functor $\mathsf{Fam}\, D \to \mathsf{Fam}\, D$. Since we are going to generalise it in a moment, we now present this functor in full. In doing so, we make use of the following folklore lemma:

Lemma 7.11 $\mathsf{Fam}\,\mathbb{C}$ is the free completion of $\mathbb{C}$ under set-indexed coproducts, i.e. $\mathsf{Fam}\,\mathbb{C}$ has set-indexed coproducts, given by

$$\sum_{a:A}(X_a, P_a) = (\sum_{a:A} X_a, [P_a]_{a:A}) \ ,$$

and there is a functor $\eta : \mathbb{C} \to \mathsf{Fam}\,\mathbb{C}$, such that every functor $F : \mathbb{C} \to \mathbb{D}$ where $\mathbb{D}$ is a category with set-indexed coproducts has a unique (up to natural isomorphism) coproduct-preserving extension $\overline{F} : \mathsf{Fam}\,\mathbb{C} \to \mathbb{D}$.

Proof. The embedding $\eta : \mathbb{C} \to \mathsf{Fam}\,\mathbb{C}$ is given by $\eta(Y) = (1, \lambda\_.\, Y)$, and the extension $\overline{F}$ of $F : \mathbb{C} \to \mathbb{D}$ by $\overline{F}(X, P) = \sum_{x:X} F(P(x))$. We have $\overline{F} \circ \eta = F$ (up to natural isomorphism) since a unary coproduct of A is nothing but the object A itself. Furthermore, $\overline{F}$ preserves coproducts since

$$\sum_{\langle a,x\rangle:\sum_{a:A} X_A} F([P_a](\langle a,x\rangle)) = \sum_{\langle a,x\rangle:\sum_{a:A} X_A} F(P_a(x)) \cong \sum_{a:A}\sum_{x:X_A} F(P_a(x)) = \sum_{a:A}\overline{F}(X_a, P_a)$$

and the same isomorphism also shows uniqueness. □

Remark 7.12 The category $\mathsf{Fam}\,\mathbb{C}$ has rich structure in other ways as well, for any category $\mathbb{C}$:

(i) $\mathsf{Fam}\,\mathbb{C}$ is fibred over Set via the split fibration $\pi(X, P) = X$. For later use, we note that a morphism $(h, k) : (X, P) \to (Y, Q)$ is a split Cartesian morphism if k is a family of identity morphisms, i.e. if $Q = P \circ h$.

(ii) $\mathsf{Fam}\,\mathbb{C}$ is cocomplete if and only if $\mathbb{C}$ has all small connected colimits [Carboni and Johnstone, 1995, dual of Prop. 2.1].

(iii) Fam is a functor $\mathsf{Cat} \to \mathsf{Cat}$; given $F : \mathbb{C} \to \mathbb{D}$, we get a functor $\mathsf{Fam}\,(F) : \mathsf{Fam}\,\mathbb{C} \to \mathsf{Fam}\,\mathbb{D}$ by composition: $\mathsf{Fam}\,(F)(X, P) = (X, F \circ P)$.

When $\mathbb{C}$ is a discrete category every morphism between families (X, P) and (Y, Q) consists only of functions $h : X \to Y$ such that $P(x) = Q(h(x))$ for all x in X. From a fibrational perspective, this amounts to the restriction to the split Cartesian fragment $\mathsf{Fam}\,|\mathbb{C}|$ of the fibration $\pi : \mathsf{Fam}\,\mathbb{C} \to \mathsf{Set}$, for $\mathbb{C}$ an arbitrary category. By restricting to this fragment, we can extend $\mathsf{Arg}_{\mathsf{IR}}(\gamma)$ and $\mathsf{Fun}_{\mathsf{IR}}(\gamma)$ to an action also on morphisms. From now on, let us write $[\![\gamma]\!](U, T)$ for $(\mathsf{Arg}_{\mathsf{IR}}(\gamma, U, T), \mathsf{Fun}_{\mathsf{IR}}(\gamma, U, T))$. The action of $[\![\gamma]\!]$ on objects was defined in Section 3.2.2. Making use of coproducts in $\mathsf{Fam}\,|\mathbb{C}|$, this action can be written

$$\begin{aligned}
[\![\iota\, c]\!](X, P) &= (\mathbf{1}, \lambda\_.\, c) \\
[\![\sigma_A\, f]\!](X, P) &= \sum_{a:A} [\![f\, a]\!](X, P) \\
[\![\delta_A\, F]\!](X, P) &= \sum_{g:A\to X} [\![F\,(P \circ g)]\!](X, P)
\end{aligned}$$

We now give the action on morphisms. Let $(h, \mathsf{id}) : (X, P) \to (Y, Q)$ be a morphism in $\mathsf{Fam}\, D$, i.e. $h : X \to Y$ and $Q \circ h = P$. We can then define

$$\begin{aligned}
[\![\iota\, c]\!](h, \mathsf{id}) &= (\mathsf{id}_{\mathbf{1}}, \mathsf{id}) \\
[\![\sigma_A\, f]\!](h, \mathsf{id}) &= [\mathsf{in}_a \circ [\![f\, a]\!](h, \mathsf{id})]_{a:A} \\
[\![\delta_A\, F]\!](h, \mathsf{id}) &= [\mathsf{in}_{h\circ g} \circ [\![F(Q \circ h \circ g)]\!](h, \mathsf{id})]_{g:A\to X}
\end{aligned}$$

Here, the last line type checks since $Q \circ h = P$, hence $Q \circ h \circ g = P \circ g$ and we can apply the induction hypothesis.

Hancock et al. [2013] introduce morphisms between IR codes. This makes $\mathsf{IR}(D)$ into a category, and the decoding $[\![-]\!] : \mathsf{IR}(D) \to [\mathsf{Fam}\, D, \mathsf{Fam}\, D]$ can be shown to be a full and faithful functor. We will draw inspiration from this in Section 7.2.2 when we generalise the semantics to endofunctors on $\mathsf{Fam}\,\mathbb{C}$ for possibly non-discrete categories $\mathbb{C}$. But first, let us look at some examples.

Example 7.13 (A universe closed under dependent sums) In Example 3.5, we saw an example of an inductive-recursive definition of a universe closed under W-types. As a variation of this example, let us consider a more modest universe containing the natural numbers and closed under Σ-types. This can also be defined using inductive-recursive definitions. Indeed, one can easily write down a code $\gamma_{\mathbb{N},\Sigma} : \mathsf{IR}(\mathsf{Set})$ for a functor that will have such a universe as its initial algebra:

$$\gamma_{\mathbb{N},\Sigma} \coloneqq \iota\, \mathbb{N} +_{\mathsf{IR}} \delta_{\mathbf{1}}(\lambda X.\, \delta_{X(\star)}(\lambda Y.\, \iota\, \Sigma(X(\star))\, Y)) \;: \mathsf{IR}(\mathsf{Set})$$

Here we have used $\gamma +_{\mathsf{IR}} \gamma' \coloneqq \sigma_{\mathbf{2}}\,(\lambda x.\, \mathsf{if}\ x\ \mathsf{then}\ \gamma\ \mathsf{else}\ \gamma')$ to encode a binary coproduct as a $\mathbf{2}$-indexed coproduct. The set $X(\star)$ is simply $X : \mathbf{1} \to \mathsf{Set}$ applied to the canonical element $\star : \mathbf{1}$. If we decode $\gamma_{\mathbb{N},\Sigma}$, we get a functor which satisfies

$$\begin{aligned}
[\![\gamma_{\mathbb{N},\Sigma}]\!](U, T) &\cong (\mathbf{1}, \lambda\_.\, \mathbb{N}) + (\Sigma u{:}U\,.\, T(u) \to U, \lambda\langle u, f\rangle.\, \Sigma\, x{:}T(u)\,.\, T(f(x))) \\
&= (\mathbf{1} + \Sigma u{:}U\,.\, T(u) \to U, \mathsf{inl}\_ \mapsto \mathbb{N}; \mathsf{inr}(u, f) \mapsto \Sigma\, x{:}T(u)\,.\, T(f(x)))
\end{aligned}$$

so that the initial algebra (U, T) of $[\![\gamma_{\mathbb{N},\Sigma}]\!]$, which satisfies $(U, T) \cong [\![\gamma_{\mathbb{N},\Sigma}]\!](U, T)$ by Lambek's Lemma, satisfies the the following equations:

$$\begin{aligned} U &= \mathbf{1} + (\Sigma u : U)(T(u) \to U) \\ T(\mathsf{inl}\ *) &= \mathbb{N} \\ T(\mathsf{inr}\ (u, f)) &= (\Sigma x : T(u))T(f(x)) \end{aligned}$$

■

Example 7.14 (A universe closed under dependent function spaces) In the same way, we can easily write a down a code for a universe closed under Π-types:

$$\gamma_{\mathbb{N},\Pi} := \iota\,\mathbb{N} +_{\mathsf{IR}} \delta_1(\lambda X.\, \delta_{X(\star)}(\lambda Y.\, \iota\,\Pi(X(\star))\,Y)) \ : \mathsf{IR}(\mathsf{Set})$$

Even though this looks extremely similar to the code in the previous example, we will see in the next section that there is a big semantic difference between them. ■

7.2.2 Syntax and semantics of positive inductive-recursive definitions

We know from the last section that IR codes can be interpreted as functors on families built over a discrete category. What happens if we try to interpret IR codes on the category Fam $\mathbb{C}$, and not just on the subcategory Fam $|\mathbb{C}|$? The problem is that if we allow for more general morphisms, we can not prove functoriality of the semantics of a δ code as it stands anymore: it is essential to have an actual equality on the second component of a morphism in Fam $\mathbb{C}$ in order to have a sound semantics (see Example 7.18 below). We now introduce a new axiomatisation of *positive inductive-recursive definitions* IR^+ which enables us to solve this problem: By generalising inductive-recursive definitions, we can interpret codes as functors on Fam $\mathbb{C}$ for an arbitrary, possibly non-discrete category $\mathbb{C}$.

The basic idea is to deploy proper functors in the δ codes. This enables us to remove the restriction on morphisms within inductive-recursive definitions; indeed, if we know that $F : (A \to \mathbb{C}) \to \mathsf{IR}^+(\mathbb{C})$ is a *functor*, and not just a *function*, we do not have to rely on the equality $P \circ g = Q \circ h \circ g$ between objects in $\mathbb{C}^A$, but we can use the second component of a morphism (h, k) in Fam $\mathbb{C}$ to get a map $P \circ g \to Q \circ h \circ g$; then we can use the fact that F is a functor to get a morphism between codes $F(P \circ g) \to F(Q \circ h \circ g)$.

However, what does it mean for $F : (A \to \mathbb{C}) \to \mathsf{IR}^+(\mathbb{C})$ to be a functor? To start with, we need both $A \to \mathbb{C}$ and $\mathsf{IR}^+(\mathbb{C})$ to be categories. It is clear that $A \to \mathbb{C}$ is just a functor category (with A a discrete category), but what about IR^+? Following Hancock et al. [2013], we can define the morphisms between IR^+ codes inductively. For ordinary inductive-recursive definitions, the morphisms can be defined after the definition of the codes themselves, but this time, the definitions needs to be done simultaneously, as we want the δ code to refer to morphisms. In other words, we are dealing with an inductive-inductive definition!

The codes will be interpreted as functors $[\![\gamma]\!] :$ Fam $\mathbb{C} \to$ Fam $\mathbb{C}$, and the morphisms as natural transformations between them. This should also give an intuition for the definition of the morphisms: a morphism between codes γ and γ' contains the data necessary to construct a natural transformation from $[\![\gamma]\!]$ to $[\![\gamma']\!]$. The actual choice of morphisms is not so important, as long as they contain identities, are closed under composition and can be decoded as natural transformations. We have made one such choice, but many others are possible.

Definition 7.15 Given a category $\mathbb{C}$ we simultaneously define the type $\mathsf{IR}^+(\mathbb{C})$ of positive inductive-recursive codes on and the type of morphisms between these codes $\mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\_,\_) : \mathsf{IR}^+(\mathbb{C}) \to \mathsf{IR}^+(\mathbb{C}) \to \mathsf{type}$ as follows:

- $\mathsf{IR}^+(\mathbb{C})$ codes:

$$\frac{c : \mathbb{C}}{\iota\, c : \mathsf{IR}^+(\mathbb{C})}$$

$$\frac{A : \mathsf{Set} \qquad f : A \to \mathsf{IR}^+(\mathbb{C})}{\sigma_A f : \mathsf{IR}^+(\mathbb{C})}$$

$$\frac{A : \mathsf{Set} \qquad F : (A \to \mathbb{C}) \to \mathsf{IR}^+(\mathbb{C}) \;\; (F \text{ functor})}{\delta_A F : \mathsf{IR}^+(\mathbb{C})}$$

- $\mathsf{IR}^+(\mathbb{C})$ morphisms:

  - identity morphisms:

$$\frac{}{\mathsf{id}_\gamma : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\gamma, \gamma)}$$

  - morphisms from ιc:

$$\frac{f : \mathsf{Hom}_{\mathbb{C}}(c, c')}{\Gamma_{\iota,\iota}(f) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\iota\, c, \iota\, c')}$$

$$\frac{a : A \qquad \rho : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\iota\, c, f(a))}{\Gamma_{\iota,\sigma}(a, \rho) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\iota\, c, \sigma_A f)}$$

$$\frac{g : A \to \mathbf{0} \qquad \rho : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\iota\, c, F(! \circ g))}{\Gamma_{\iota,\delta}(g, \rho) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\iota\, c, \delta_A F)}$$

  - morphisms from $\sigma_A f$:

$$\frac{\gamma, : \mathsf{IR}^+(\mathbb{C}) \qquad \rho : (a : A) \to \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(f(a), \gamma)}{\Gamma_{\sigma,\gamma}(\rho) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\sigma_A f, \gamma)}$$

  - morphisms from $\delta_A F$

$$\frac{\gamma, : \mathsf{IR}^+(\mathbb{C}) \qquad \rho : \mathsf{Nat}(F, \kappa_\gamma)}{\Gamma_{\delta,\gamma}(\rho) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\delta_A F, \gamma)}$$

$$\frac{b : B \qquad \rho : \mathsf{Nat}(F, \kappa_{f\,b})}{\Gamma_{\delta,\sigma}(b, \rho) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\delta_A F, \sigma_A f)}$$

$$\frac{g : B \to A \qquad \rho : \mathsf{Nat}(F, G(- \circ g))}{\Gamma_{\delta,\delta}(g, \rho) : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\delta_A F, \delta_B G)}$$

In the clauses $\Gamma_{\delta,\gamma}$ and $\Gamma_{\delta,\sigma}$, we have written $\kappa_\gamma : \mathbb{C}^A \to \mathsf{IR}^+(\mathbb{C})$ for the constant functor with value γ. ■

This is a (large) inductive-inductive definition of

$$\begin{aligned} \mathsf{IR}^+(\mathbb{C}) &: \mathsf{type} \\ \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\_,\_) &: \mathsf{IR}^+(\mathbb{C}) \to \mathsf{IR}^+(\mathbb{C}) \to \mathsf{type} \end{aligned}$$

Since $\mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\_,\_)$ is indexed over two copies of $\mathsf{IR}^+(\mathbb{C})$, it is an instance of the extended theory from Section 6.2. The simultaneousness is hidden in the demand that F should be a functor in the δ code. Indeed, this can be spelt out as two operations

$$\begin{aligned} &F : (A \to \mathbb{C}) \to \mathsf{IR}^+(\mathbb{C}) \\ &F_\to : (f, g : A \to \mathbb{C}) \to \mathsf{Hom}_{A \to \mathbb{C}}(f, g) \to \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(F(f), F(g)) \end{aligned}$$

plus the functor laws, which we can leave out of the initial definition. Note how the type of $F_\to$ indeed is strictly positive in $\mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}$.

We now explain how each code $\gamma : \mathsf{IR}^+(\mathbb{C})$ is interpreted as an endofunctor

$$[\![\gamma]\!] : \mathsf{Fam}\,\mathbb{C} \to \mathsf{Fam}\,\mathbb{C}$$

Let us call a functor which is isomorphic to a functor induced by an IR^+ code an IR^+ functor. The semantics of IR^+ closely follows the one given in Section 7.2.1; as before we make essential use of coproducts in $\mathsf{Fam}\,\mathbb{C}$. However, since codes and morphisms were defined simultaneously, they also need to be decoded simultaneously as functors and natural transformations respectively. This is exactly what the inductive-inductive elimination principle allows us to do!

Theorem 7.16 (IR^+ functors) Let $\mathbb{C}$ be an arbitrary category.

(i) Every code $\gamma : \mathsf{IR}^+(\mathbb{C})$ induces a functor $[\![\gamma]\!] : \mathsf{Fam}\,\mathbb{C} \to \mathsf{Fam}\,\mathbb{C}$.

(ii) Every morphism $\rho : \mathsf{Hom}_{\mathsf{IR}^+(\mathbb{C})}(\gamma, \gamma')$ for codes $\gamma, \gamma' : \mathsf{IR}^+(\mathbb{C})$ gives rise to a natural transformation $[\![\rho]\!] : [\![\gamma]\!] \overset{\cdot}{\longrightarrow} [\![\gamma']\!]$.

Proof. While the action on objects is the same for both IR^+ and IR functors, the action on morphisms is different when interpreting a code of type $\delta_A F$: in the semantics of IR^+ we exploit the fact that $F : (A \to \mathbb{C}) \to \mathsf{IR}^+(\mathbb{C})$ is now a functor by using its action on morphism (which we, for the sake of clarity, indicate with $F_{\rightarrow}$). We give the action of IR^{+} functors on morphisms only.

The action on morphisms is given as follows. Let $(h,k) : (X,P) \to (Y,Q)$ be a morphism in $\mathsf{Fam}\,\mathbb{C}$. We define $\llbracket\gamma\rrbracket(h,k) : \llbracket\gamma\rrbracket(X,P) \to \llbracket\gamma\rrbracket(Y,Q)$ by recursion on γ:

$$\begin{aligned}
\llbracket\iota\, c\rrbracket(h,k) &= (\mathsf{id}_{\mathbf{1}}, \mathsf{id}_c) \\
\llbracket\sigma_A f\rrbracket(h,k) &= [\mathsf{in}_a \circ \llbracket f\, a\rrbracket(h,k)]_{a:A} \\
\llbracket\delta_A F\rrbracket(h,k) &= [\mathsf{in}_{h\circ g} \circ \llbracket F(Q\circ h\circ g)\rrbracket(h,k) \circ \llbracket F_{\rightarrow}(g^{*}(k))\rrbracket_{(X,P)}]_{g:A\to X}
\end{aligned}$$

In the last clause, $g^{*}(k) : P\circ g \overset{\cdot}{\longrightarrow} Q\circ h\circ g$ is the natural transformation with component $g^{*}(k)_a = k_{g\,a} : P(g\,a) \to Q(k(g\,a))$.

We now explain how a IR^{+} morphism $\rho : \gamma \to \gamma'$ is interpreted as natural transformation $\llbracket\rho\rrbracket : \llbracket\gamma\rrbracket \overset{\cdot}{\longrightarrow} \llbracket\gamma'\rrbracket$ between IR^{+} functors by specifying the component $\llbracket\rho\rrbracket_{(X,P)}$ at $(X,P) : \mathsf{Fam}\,\mathbb{C}$. Naturality of these transformations can be proved by a routine diagram chase.

$$\begin{aligned}
\llbracket\mathsf{id}_{\gamma}\rrbracket_{(X,P)} &= \mathsf{id}_{\llbracket\gamma\rrbracket(X,P)} \\
\llbracket\Gamma_{\iota,\iota}(f)\rrbracket_{(X,P)} &= (\mathsf{id}_{\mathbf{1}}, f) \\
\llbracket\Gamma_{\iota,\sigma}(a,\rho)\rrbracket_{(X,P)} &= \mathsf{in}_a \circ \llbracket\rho\rrbracket_{(X,P)} \\
\llbracket\Gamma_{\iota,\delta}(g,\rho)\rrbracket_{(X,P)} &= \mathsf{in}_{!_X\circ g} \circ \llbracket\rho\rrbracket_{(X,P)} \\
\llbracket\Gamma_{\sigma,\gamma}(\rho)\rrbracket_{(X,P)} &= [\llbracket\rho(a)\rrbracket_{(X,P)}]_{a:A} \\
\llbracket\Gamma_{\delta,\gamma}(\rho)\rrbracket_{(X,P)} &= [\llbracket\rho_{(P\circ h)}\rrbracket_{(X,P)}]_{h:A\to X} \\
\llbracket\Gamma_{\delta,\sigma}(b,\rho)\rrbracket_{(X,P)} &= \mathsf{in}_b \circ [\llbracket\rho_{(P\circ g)}\rrbracket_{(X,P)}]_{g:A\to X} \\
\llbracket\Gamma_{\delta,\delta}(f,\rho)\rrbracket_{(X,P)} &= [\mathsf{in}_{g\circ f} \circ \llbracket\rho_{(P\circ g)}\rrbracket_{(X,P)}]_{g:A\to X}
\end{aligned}$$

□

Formally, we are applying the elimination principle to the motive

$$\begin{aligned}
P(\gamma) &= \mathsf{Fam}\,\mathbb{C} \to \mathsf{Fam}\,\mathbb{C} \\
Q(\gamma,\gamma',\rho,\widetilde{\gamma},\widetilde{\gamma'}) &= (X : \mathsf{Fam}\,\mathbb{C}) \to \widetilde{\gamma}(X) \to \widetilde{\gamma'}(X)
\end{aligned}$$

Note how it is once again crucial that Q can depend on $\widetilde{\gamma}$ and $\widetilde{\gamma'}$, since we need to define functors mutually with natural transformations between them.

Example 7.17 (A universe closed under dependent sums in $\mathsf{Fam}\,\mathsf{Set}^{\mathsf{op}}$) In Example 7.13, we defined an ordinary IR code $\gamma_{\mathbb{N},\Sigma} : \mathsf{IR}(\mathsf{Set})$ for a universe closed under Σ-types. We can extend this code to an IR^{+} code

$$\gamma_{\mathbb{N},\Sigma} = \iota\,\mathbb{N} +_{\mathsf{IR}} \delta_{\mathbf{1}}(X \mapsto \delta_{X*}(Y \mapsto \iota\,\Sigma(X*)\,Y)) : \mathsf{IR}^{+}(\mathsf{Set}^{\mathsf{op}})$$

where now $G := Y \mapsto \iota\,\Sigma(X*)\,Y$ and $F := X \mapsto \delta_{X*}\,G$ needs to be functors. Given $f : Y \to Y'$ in $X \to \mathsf{Set}^{\mathsf{op}}$, i.e. $f_x : Y(x) \to Y'(x)$ in $\mathsf{Set}^{\mathsf{op}}$, we have $\Sigma x : (X*).f_x : \Sigma(X*)\,Y \to \iota\,\Sigma(X*)\,Y'$ in $\mathsf{Set}^{\mathsf{op}}$ so that we can define

$$G(f) : \iota\,\Sigma(X*)\,Y \to \iota\,\Sigma(X*)\,Y'$$

by $G(f) = \Gamma_{\iota,\iota}(\Sigma x : (X*).f_x)$.

We also need F to be a functor. Given $f : X \to X'$ in $\mathbf{1} \to \mathsf{Set}^{\mathsf{op}}$, we need to define $F(f) : \delta_{X(*)}\, G \to \delta_{X'(*)}\, G$. According to Definition 7.15, such a morphism consists of a map $g : X'(*) \to X(*)$ and a natural transformation ρ from G to $G(- \circ g)$. We can choose $g = f_* : X'(*) \to X(*)$ and $\rho = [\mathsf{in}_{f_* x}]_{x:X'(*)}$. Notice that working in $\mathsf{Set}^{\mathsf{op}}$ made sure that f_* was going in the right direction. ■

Example 7.18 (A universe closed under dependent function spaces in $\mathsf{Fam}\ \mathsf{Set}^{\cong}$) In Example 7.14, we saw how we could use induction-recursion to define a universe closed under Π-types in $\mathsf{Fam}\ |\mathsf{Set}|$, using the following code:

$$\gamma_{\mathbb{N},\Pi} = \iota\,\mathbb{N} +_{\mathsf{IR}} \delta_1(X \mapsto \delta_{X*}(Y \mapsto \iota\,\Pi(X*)\,Y)) \ : \mathsf{IR}(\mathsf{Set})$$

If we try to extend this to an IR^+ code in $\mathsf{Fam}\ \mathsf{Set}$ or $\mathsf{Fam}\,\mathsf{Set}^{\mathsf{op}}$, we run into problems. Basically, given a morphism $f : X' \to X$, we need to construct a morphism $\Pi\, X'\, (Y \circ f) \to \Pi\, X\, Y$, which of course is impossible if e.g. $X' = \mathbf{0}$, $X = \mathbf{1}$, and $Y* = \mathbf{0}$.

Hence the inherent contravariance in the Π-type means that $\gamma_{\mathbb{N},\Pi}$ does not extend to a $\mathsf{IR}^+(\mathsf{Set})$ or $\mathsf{IR}^+(\mathsf{Set}^{\mathsf{op}})$ code. However, if we move to the groupoid $\mathsf{Set}^{\cong}$, which is the subcategory of Set with only isomorphisms as morphisms, we do get an $\mathsf{IR}^+(\mathsf{Set}^{\cong})$ code describing the universe in question, which is still living in a category beyond the strict category $\mathsf{Fam}\ |\mathsf{Set}|$. ■

7.2.3 Comparison to plain IR

We now investigate the relationship between IR^+ and IR. Note that every type D can be regarded as a discrete category, which we by abuse of notation denote $|D|$. In the other direction, every category $\mathbb{C}$ gives rise to a type $|\mathbb{C}|$ whose elements are the objects of $\mathbb{C}$.

Proposition 7.19 There is a function $\varphi : \mathsf{IR}(D) \to \mathsf{IR}^+(|D|)$ s.t.

$$[\![\gamma]\!]_{\mathsf{IR}(D)} \cong [\![\varphi(\gamma)]\!]_{\mathsf{IR}^+(|D|)}$$

Proof. The only interesting case is the δ code. Since $|D|$ is a discrete category, also $A \to |D|$ is discrete. Hence a mapping on objects $(A \to |D|) \to \mathsf{IR}(D)$ can trivially be extended to a functor $(A \to |D|) \to \mathsf{IR}^+(|D|)$. □

This proposition shows that the theory of IR can be embedded in the theory of IR^+. In the next proposition we slightly sharpen this result. We use the functoriality of the Fam construction (Remark 7.12) to show that forgetting about the extra structure in IR^+ simply gets us back to plain IR.

Proposition 7.20 Let $|-| : \mathsf{Cat} \to \mathsf{Set}$ be the functor assigning to each category the collection of its objects. There is a function $\psi : \mathsf{IR}^+\mathbb{C} \to \mathsf{IR}\,|\mathbb{C}|$ such that

$$\mathsf{Fam}\ |-| \circ [\![\gamma]\!]_{\mathsf{IR}^+\mathbb{C}} = [\![\psi(\gamma)]\!]_{\mathsf{IR}\,|\mathbb{C}|} \circ \mathsf{Fam}\ |-|$$

for all $\gamma : \mathsf{IR}^+\mathbb{C}$. Furthermore, $\psi \circ \varphi = \mathsf{id}$. □

7.2.4 Existence of initial algebras

We now generalise Dybjer and Setzer's model construction from 6.1.3.1. Inspecting the proof, we see that it indeed is possible to adapt it also for the more general setting of positive inductive-recursive definitions by making the appropriate adjustments.

We call a morphism $(h, k) : (U, T) \to (U, T')$ in $\mathsf{Fam}\,\mathbb{C}$ a *splitting morphism* if $k = \mathsf{id}_T$, i.e. $T' \circ h = T$, since these are the chosen morphisms in the split subfibration $\pi : \mathsf{Fam}\,|\mathbb{C}| \to \mathsf{Set}$. In other words, we write $\mathsf{Fam}\,|\mathbb{C}|$ for the category with the same objects as $\mathsf{Fam}\,\mathbb{C}$, but with splitting morphisms only.

Inspecting the proofs in Section 6.1.3.1, we see that they crucially depend on morphisms being splitting in several places. Luckily, the morphisms involved in the corresponding proofs for IR^+ actually are! We show that the initial chain of a IR^+ functor actually lives in $\mathsf{Fam}\,|\mathbb{C}|$, which will allow us to modify Dybjer and Setzer's proof accordingly.

Lemma 7.21 For every code $\gamma : \mathsf{IR}^+\,\mathbb{C}$ the induced functor $[\![\gamma]\!] : \mathsf{Fam}\,\mathbb{C} \to \mathsf{Fam}\,\mathbb{C}$ preserves splitting morphisms, i.e. if (f, g) is splitting, then so is $[\![\gamma]\!](f, g)$.

Proof. By induction on the structure of the code. The interesting case is $\gamma = \delta_A F$. Let $(h, \mathsf{id}) : (X, P \circ h) \to (Y, P)$ be a splitting morphism. We have

$$\begin{aligned} [\![\delta_A F]\!](h, \mathsf{id}) &= [\mathsf{in}_{h\circ g} \circ [\![F(P \circ h \circ g)]\!](h, \mathsf{id}) \circ [\![F_{\to}(g^*(\mathsf{id}))]\!]_{(X,P)}]_{g:A\to X} \\ &= [\mathsf{in}_{h\circ g} \circ [\![F(P \circ h \circ g)]\!](h, \mathsf{id})]_{g:A\to X} \end{aligned}$$

where $[\![F(g^*\mathsf{id})]\!]_{(X,P)} = \mathsf{id}$ since both g^*, F and $[\![\_]\!]$ are functors. By the induction hypothesis, each $[\![F(P \circ h \circ g)]\!](h, \mathsf{id})$ is splitting. Furthermore injections are splitting in $\mathsf{Fam}\,\mathbb{C}$. Since composition of splitting morphisms is still splitting and the cotuple of splitting morphisms is also splitting in $\mathsf{Fam}\,\mathbb{C}$ we conclude that $[\![\delta_A\, F]\!](h, \mathsf{id})$ is a splitting morphism. □

Lemma 7.22 For each $\gamma : \mathsf{IR}^+\,\mathbb{C}$, the initial chain

$$0 \to [\![\gamma]\!](0) \to [\![\gamma]\!]^2(0) \to \dots$$

consists of splitting morphisms only.

Proof. Recall that the connecting morphisms $\omega_{j,k} : [\![\gamma]\!]^j(0) \to [\![\gamma]\!]^k(0)$ are uniquely determined as follows:

- $\omega_{0,1} = !_{[\![\gamma]\!](0)}$ is unique.
- $\omega_{j+1,k+1}$ is $[\![\gamma]\!](\omega_{j,k}) : [\![\gamma]\!]([\![\gamma]\!]^j(0)) \to [\![\gamma]\!]([\![\gamma]\!]^k(0))$.
- $\omega_{j,k}$ is the colimit cocone for j a limit ordinal.

We prove the statement by induction on j. It is certainly true that $!_{[\![\gamma]\!](0)} : (0, !) \to [\![\gamma]\!](0)$ is an identity at each component – there are none. Thus $\omega_{0,1}$ is a splitting morphism. At successor stages, we can directly apply Lemma 7.21 and the induction hypothesis.

Finally, at limit stages, we use the fact that the colimit lives in Fam $|\mathbb{C}|$ and hence coincides with the colimit in that category on splitting morphisms, so that the colimit cocone is splitting. □

Inspecting Dybjer and Setzer's original proof, we see that it now goes through also for IR^+ if we insert appeals to Lemma 7.22 where necessary. To finish the proof, we also need to ensure that Fam $\mathbb{C}$ has κ-filtered colimits; this is automatically true if $\mathbb{C}$ has all small connected colimits (compare Remark 7.12), since Fam $\mathbb{C}$ then is cocomplete. Note that discrete categories have all small connected colimits for trivial reasons.

Theorem 7.23 Assume that a Mahlo cardinal exists in the meta-theory. If $\mathbb{C}$ has connected colimits, then every functor $[\![\gamma]\!]$ for $\gamma : \mathsf{IR}^+\,\mathbb{C}$ has an initial algebra. □

7.2.5 Conclusion

We have introduced the theory IR^+ of positive inductive-recursive definitions as a generalisation of inductive-recursive definitions IR. Crucial for the definition of IR^+, where codes and morphisms between codes are defined simultaneously, is having access to inductive-inductive definitions in the metatheory. We saw further examples of this when we defined the semantics of IR^+, where the general elimination rules of inductive-inductive definitions were needed to simultaneously interpret codes as functors and morphisms between codes as natural transformations. The theory IR^+, with IR as a subtheory, hints at the possibility of a more sophisticated analysis of inductive-recursive data types, where not only a type U and a decoding function $T : U \to D$ are introduced, but also the intrinsic structure between objects in the target type D is taken into account. Such structure exists for example when D is a setoid, the category Set or $\mathrm{Set}^{\mathrm{op}}$, a groupoid or, in general, an arbitrary category $\mathbb{C}$.

Chapter 8

Conclusions

Contents

In this final chapter, we summarise the content of the thesis and discuss future work.

8.1 Summary and discussion

This thesis claims that

> *Advanced forms of inductive definitions are important both for programming and proving in Martin-Löf Type Theory.*

In support of this claim, we have studied the class of *inductive-inductive definitions*, mostly from a theoretical perspective, but also by concretely exploring problems where such definitions play a crucial rôle. In more detail, we have:

- Given a finite axiomatisation of inductive-inductive definitions, which generalises axiomatisations of ordinary inductive definitions and indexed inductive definitions. We argue that this is a natural way to extend a base type theory with a universe of data types.
- Shown how inductive-inductive definitions can be characterised categorically as initial objects in a certain category of dialgebras. This shows that inductive-inductive definitions have an interesting, mathematically well-behaved structure. This is important for establishing meta-theoretical properties of the system of data types.
- Modelled inductive-inductive definitions in a straightforward way in set theory. The model is proof-theoretically wasteful, but conceptually fitting, as it shows

that a "naïve" mental understanding of inductive-inductive definitions (types are sets, terms of function type are set-theoretical functions, ...) is possible.

- Translated a version of inductive-inductive definitions with restricted elimination rules into the theory of indexed inductive definitions. This shows that the proof-theoretical strength of this version of inductive-inductive definitions coincides with the strength of indexed inductive definitions, while the general theory has the same strength as indexed inductive definitions with "recursive-recursive" elimination rules.

- Investigated various extensions of inductive-inductive definitions, such as the definition of generalised families $A : \mathsf{Set}$ and $B : F(A) \to \mathsf{Set}$, higher towers of inductive-inductive definitions

$$A : \mathsf{Set} \qquad B : A \to \mathsf{Set} \qquad C : (x : A) \to B(x) \to \mathsf{Set}$$

  and a proof-theoretically strong combination of inductive-inductive and inductive-recursive definitions. This shows that the theory is adaptable, and also able to accommodate to the forms of inductive-inductive definitions that actually occur in the literature.

- Finally, we explored two larger case studies where inductive-inductive definitions were used to develop actual mathematics, supporting the thesis that advanced data types can be very helpful for mathematics in a type-theoretical setting.

8.2 Further work

Let discuss some interesting topics for further research:

Internal fixed points Our axiomatisation only allows *direct* inductive arguments, i.e. arguments of the form $K \to A$, and not of the form $K \to \mathsf{List}(A)$. Following Morris et al. [2009], we could hope to support the latter by adding *internal fixed points* to our theory. Morris et al. do this for indexed inductive definitions; it is unclear how easy it would be to extend their system to cover also inductive-inductive definitions. For the case of such nested definitions in the proof assistant Minlog (working with simple types and ordinary first order logic), see Miyamoto, Nordvall Forsberg, and Schwichtenberg [2013].

Coinductive-coinductive definitions In this thesis, we have focused exclusively on inductive definitions, initial algebras and least fixed points. It is natural to ask how much of the current work carries over to coinductive definitions, terminal algebras and greatest fixed points. Is there a theory of coinductive-coinductive definitions? What about letting the index set A be inductive, but the family $B : A \to \mathsf{Set}$ coinductive or vice versa, leading to inductive-coinductive or coinductive-inductive definitions? Are such theories useful? It seems easier to make sense of these questions compared to the corresponding questions for inductive-recursive definition, where there is a tighter interplay between the definition of the two components. Recently Capretta [2013] has made progress in this direction for small inductive-recursive definitions.

An abstract framework Is there a uniform framework which allows general combinations of inductive-inductive and inductive-recursive definitions? Our recent work on fibrational presentations of inductive-recursive definitions [Ghani, Malatesta, Nordvall Forsberg, and Setzer, 2013b] seems to be a promising starting point. This might make it possible to e.g. allow also a second recursively defined function

$$T' : (x : A) \to B(x) \to T(x) \to D \ ,$$

while at the same time reduce the syntactical complexity of the theory.

Implementations It would be interesting to see what it would take to actually implement the theories presented in this thesis, especially the extensions from Chapter 6. I conjecture that the axiomatisation in Chapter 3 would be relatively simple to implement – all that is needed is the addition of some constants and some reductions to a standard implementation of Type Theory. One would probably end up with something similar to the implementation of data types in Epigram 2 [Chapman et al., 2010]. It remains to be seen if it is possible to make the codes for inductive-inductive definitions levitate as well.

Going beyond inductive-inductive and inductive-recursive definitions Finally, in the proof-theoretical landscape, inductive-inductive, and even inductive-recursive definitions are quite tame. Anton Setzer has invented several universes that go beyond inductive-recursive definitions [Setzer, 2008; Kahle and Setzer, 2010], but are still consistent and (arguably) constructively justified. Is there a theory of data types that contain these wilder examples?

APPENDIX A

Agda formalisations

This appendix contains examples of inductive-inductive data types in Agda, as well as Agda formalisations of the axiomatisations of inductive-inductive definitions (from Section 3.2.3), inductive-recursive definitions (from Section 3.2.2) and inductive-inductive-recursive definitions (from Section 6.1).

All code type check with Agda 2.3.2.2 and the standard library version 0.7.

A.1 Examples

We give Agda implementations of some inductive-inductive definitions considered in this thesis.

A.1.1 Contexts and types and terms

The contexts and types and terms from Examples 3.1 and 6.1.

```
module contexts-types-terms where

mutual
  data Ctxt : Set where
    ε   : Ctxt
    _::_ : (Γ : Ctxt) -> Ty Γ -> Ctxt

  data Ty : Ctxt -> Set where
    ι    : (Γ : Ctxt) -> Ty Γ
    ‘Set : (Γ : Ctxt) -> Ty Γ
    El   : (Γ : Ctxt) -> Tm Γ (‘Set Γ) -> Ty Γ
    Pi   : (Γ : Ctxt) -> (A : Ty Γ) -> Ty (Γ :: A) -> Ty Γ
    Wk   : (Γ : Ctxt) -> (A : Ty Γ) -> Ty Γ -> Ty (Γ :: A)

  data Tm : (Γ : Ctxt) -> Ty Γ -> Set where
    top : (Γ : Ctxt) -> (A : Ty Γ) -> Tm (Γ :: A) (Wk Γ A A)
    wk  : (Γ : Ctxt) -> (A : Ty Γ) -> (B : Ty Γ) -> Tm Γ B
```

```
                                                   -> Tm (Γ :: A) (Wk Γ A B)
    lam : (Γ : Ctxt) -> (A : Ty Γ) -> (B : Ty (Γ :: A)) -> Tm (Γ :: A) B
                                                         -> Tm Γ (Pi Γ A B)
```

A.1.2 Sorted lists

Sorted lists from Example 3.2, together with the insert function from Section 3.2.5.1.

```
module sortedList where

open import Data.Nat
open import Relation.Nullary
open import Relation.Binary
open import Data.Empty
open import Data.Unit using (⊤)

-------------------------------------------------------------------------
-- Introduction rules
-------------------------------------------------------------------------
mutual
  data SList : Set where
    [] : SList
    _::_⟨_ : (x : ℕ) -> (ys : SList) -> x ≤L ys → SList

  data _≤L_ : (n : ℕ) -> SList -> Set where
    triv : {n : ℕ} -> n ≤L []
    cons : {m : ℕ} -> {n : ℕ} -> {ys : SList} -> {p : n ≤L ys} ->
           (m ≤ n) -> m ≤L ys -> m ≤L (n :: ys ⟨ p)
-------------------------------------------------------------------------


-------------------------------------------------------------------------
-- An example sorted list
-------------------------------------------------------------------------
private
  ex : SList
  ex = 0 :: 1 :: 2 :: 3 :: [] ⟨ triv
                               ⟨ cons (s≤s (s≤s z≤n)) triv
                                ⟨ cons (s≤s z≤n) (cons (s≤s z≤n) triv)
                                 ⟨ cons z≤n (cons z≤n (cons z≤n triv))
-------------------------------------------------------------------------


-------------------------------------------------------------------------
-- Elimination rules
-------------------------------------------------------------------------
mutual
  elimSList : (P : SList -> Set)
              (Q : (n : ℕ) -> (ys : SList) -> n ≤L ys -> P ys -> Set) ->
              (step[] : P [])
              (step:: : (n : ℕ) -> (ys : SList) -> (p : n ≤L ys) ->
                        (pp : P ys) -> Q n ys p pp -> P (n :: ys ⟨ p)) ->
              (steptriv : (n : ℕ) -> Q n [] triv step[])
              (stepcons : (m : ℕ) -> {n : ℕ} -> {ys : SList} ->
                          {p : n ≤L ys} -> (m<n : m ≤ n) -> (p' : m ≤L ys) ->
                          (pp : P ys) -> (qq : Q n ys p pp) ->
                          (qqq : Q m ys p' pp)
                            -> Q m (n :: ys ⟨ p) (cons m<n p')
                                                 (step:: n ys p pp qq))
              (ys : SList) -> P ys
  elimSList P Q step[] step:: steptriv stepcons [] = step[]
  elimSList P Q step[] step:: steptriv stepcons (n :: ys ⟨ p)
     = step:: n ys p (elimSList P Q step[] step:: steptriv stepcons ys)
                     (elim≤ P Q step[] step:: steptriv stepcons n ys p)

  elim≤ : (P : SList -> Set)
          (Q : (n : ℕ) -> (ys : SList) -> n ≤L ys -> P ys -> Set) ->
          (step[] : P [])
          (step:: : (n : ℕ) -> (ys : SList) -> (p : n ≤L ys) ->
                    (pp : P ys) -> Q n ys p pp -> P (n :: ys ⟨ p)) ->
          (steptriv : (n : ℕ) -> Q n [] triv step[])
          (stepcons : (m : ℕ) -> {n : ℕ} -> {ys : SList} ->
                      {p : n ≤L ys} -> (m<n : m ≤ n) -> (p' : m ≤L ys) ->
                      (pp : P ys) -> (qq : Q n ys p pp) ->
                      (qqq : Q m ys p' pp)
                        -> Q m (n :: ys ⟨ p) (cons m<n p')
                                             (step:: n ys p pp qq))
          (n : ℕ) -> (ys : SList) -> (p : n ≤L ys)
           -> Q n ys p (elimSList P Q step[] step:: steptriv stepcons ys)
  elim≤ P Q step[] step:: steptriv stepcons m [] triv = steptriv m
  elim≤ P Q step[] step:: steptriv stepcons m (n :: ys ⟨ p) (cons q p')
    = stepcons m q p' (elimSList P Q step[] step:: steptriv stepcons ys)
                      (elim≤ P Q step[] step:: steptriv stepcons n ys p)
                      (elim≤ P Q step[] step:: steptriv stepcons m ys p')
------------------------------------------------------------------------

------------------------------------------------------------------------
-- Some lemmas about ≤ and ≤L
------------------------------------------------------------------------
trans-≤ : ∀ {k m n} -> k ≤ m -> m ≤ n -> k ≤ n
trans-≤ = IsDecTotalOrder.trans
            (DecTotalOrder.isDecTotalOrder Data.Nat.decTotalOrder)

≤L-trans : ∀ {x y} -> (zs : SList) -> x ≤ y -> y ≤L zs -> x ≤L zs
≤L-trans [] x<y all = triv
≤L-trans (y' :: ys ⟨ p) x<y (cons y<y' y<ys)
  = cons (trans-≤ x<y y<y') (≤L-trans ys x<y y<ys)

¬x<y→y<x : {x y : ℕ} -> (x ≤ y -> ⊥) -> y ≤ x
¬x<y→y<x {zero}  p = ⊥-elim (p z≤n)
¬x<y→y<x {y = zero} p = z≤n
```

```
¬x<y→y<x {suc n} {suc m} p = s≤s (¬x<y→y<x (λ x → p (s≤s x)))
--------------------------------------------------------------------------


--------------------------------------------------------------------------
-- Insert (defined using pattern matching for simplicity)
--------------------------------------------------------------------------
mutual
  insert : (m : ℕ) -> SList -> SList
  insert m [] = m :: [] ⟨ triv
  insert m (n :: ys ⟨ p) with m ≤? n
  ... | yes q =  m :: (n :: ys ⟨ p) ⟨ (cons q (≤L-trans ys q p))
  ... | no ¬q = n :: (insert m ys) ⟨ lemma (¬x<y→y<x ¬q) p

  lemma : ∀ {x y ys} -> y ≤ x -> y ≤L ys -> y ≤L (insert x ys)
  lemma {ys = []} y≤x y≤ys = cons y≤x y≤ys
  lemma {x} {y} {y' :: ys ⟨ p} y≤x (cons y≤y' y≤ys) with x ≤? y'
  ... | yes q = cons y≤x (cons y≤y' y≤ys)
  ... | no ¬q = cons y≤y' (lemma {x} {y} {ys} y≤x y≤ys)
--------------------------------------------------------------------------


--------------------------------------------------------------------------
-- We can also forget that a sorted list is sorted
--------------------------------------------------------------------------
open import Data.List

forget : SList -> List ℕ
forget [] = []
forget (x :: ys ⟨ y) = x :: (forget ys)
```

Insertion sort

We can use insert to define insertsort, and then define a data type of list permutations as alluded to in the introduction and prove insertsort correct:

```
open import Data.Product
open import Relation.Binary.PropositionalEquality
  renaming (trans to trans-≡; sym to sym-≡)
open import Data.List.Any hiding (tail)
open Membership-≡

--------------------------------------------------------------------------
-- Permutations of lists
--------------------------------------------------------------------------
data Permutation {A : Set} : List A -> List A -> Set where
  refl : {ys : List A} -> Permutation ys ys
  head : {x y  : A}{xs ys : List A} -> Permutation xs ys
                                   -> Permutation (x :: y :: xs) (y :: x :: ys)
  tail : {x : A}{xs ys : List A} -> Permutation xs ys
                                      -> Permutation (x :: xs) (x :: ys)
  trans : {xs ys zs : List A} -> Permutation xs ys -> Permutation ys zs
                                                          -> Permutation xs zs

-- Permutations are equivalence relations
sym : {A : Set} -> {xs ys : List A} -> Permutation xs ys -> Permutation ys xs
sym refl = refl
sym (head p) = head (sym p)
sym (tail p) = tail (sym p)
sym (trans p q) = trans (sym q) (sym p)

perm-setoid : {A : Set} -> Setoid _ _
perm-setoid {A}  = record { Carrier = List A;
                            _≈_ = Permutation;
                            isEquivalence = record { refl = refl;
                                                     sym = sym;
                                                     trans = trans } }
-------------------------------------------------------------------------

-------------------------------------------------------------------------
-- Sanity check: the definition makes sense
-------------------------------------------------------------------------
perm-correct : {A : Set}{xs ys : List A} -> Permutation xs ys -> xs ⊆ ys
perm-correct refl q = q
perm-correct (head p) (here px) = there (here px)
perm-correct (head p) (there (here px)) = here px
perm-correct (head p) (there (there r)) = there (there (perm-correct p r))
perm-correct (tail p) (here px) = here px
perm-correct (tail p) (there r) = there (perm-correct p r)
perm-correct (trans p q) r = perm-correct q (perm-correct p r)

lemma-perm-length : {A : Set}{xs ys : List A} -> Permutation xs ys
                                                      -> length xs ≡ length ys
lemma-perm-length refl = refl
lemma-perm-length (head p) rewrite lemma-perm-length p = refl
lemma-perm-length (tail p) rewrite lemma-perm-length p = refl
lemma-perm-length (trans p q) rewrite lemma-perm-length p
                                       | lemma-perm-length q = refl
-------------------------------------------------------------------------


-- Permutations of an ordinary and a sorted list
Permutation' : List ℕ -> SList -> Set
Permutation' xs ys = Permutation xs (forget ys)

-- Peano's fourth axiom, needed below to rule out the possibility of a
-- permutation between an empty and a non-empty list (not obvious because
-- of trans)
Peano-four : {n : ℕ} -> zero ≡ suc n -> ⊥
Peano-four q = subst (λ { zero -> ℕ ; (suc n) -> ⊥}) q 0
-------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
-- Inserting an element preserves permutations
-----------------------------------------------------------------------------
lemma-insert : ∀ {x} xs ys -> Permutation’ xs ys
                     -> Permutation’ (x :: xs) (insert x ys)
lemma-insert [] [] p = refl
lemma-insert [] (y :: ys ⟨ p) q  = ⊥-elim (Peano-four (lemma-perm-length q))
lemma-insert (x :: xs) [] q = ⊥-elim (Peano-four (sym-≡ (lemma-perm-length q)))
lemma-insert {z} (x :: xs) (y :: ys ⟨ p) w with z ≤? y
... | yes _ = tail w
... | no _  = begin
                z :: x :: xs
              ≈⟨ tail w ⟩
                z :: y :: forget ys
              ≈⟨ head refl ⟩
                y :: z :: forget ys
              ≈⟨ tail (lemma-insert (forget ys) ys refl) ⟩
                y :: forget (insert z ys)
              ∎
              where open import Relation.Binary.EqReasoning (perm-setoid {ℕ})
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- Insertion sort, with correctness proof
-----------------------------------------------------------------------------
insertsort : (xs : List ℕ) -> Σ[ ys ∈ SList ] Permutation’ xs ys
insertsort [] = [] , refl
insertsort (x :: xs) = insert x (proj₁ (insertsort xs)) ,
                      lemma-insert xs _ (proj₂ (insertsort xs))
```

A.1.3 Dense completion of an ordered set

The dense completion of an ordered set from Example 3.3.

```
module dense (S : Set) (_<_ : S -> S -> Set) where

  open import Data.Product

  mutual
    data S* : Set where
      η : S -> S*
      mid : (s t : S*) -> s <* t -> S*

    data _<*_ : S* -> S* -> Set where
      η< : {s t : S} -> s < t -> η s <* η t
      midl : {s t : S*} -> (p : s <* t) -> mid s t p <* t
      midr : {s t : S*} -> (p : s <* t) -> s <* mid s t p


  module universal-property
```

```
         (S' : Set)(_<'_ : S' -> S' -> Set)
         (<'-dense : {x y : S'} -> x <' y -> Σ[ z ∈ S' ] x <' z × z <' y)
         (f : S -> S')(f< : {x y : S} -> x < y -> f x <' f y) where

    mutual
      h : S* -> S'
      h (η s) = f s
      h (mid s t p) = proj₁ (<'-dense {h s} {h t} (h< p))

      h< : {s t : S*} -> s <* t -> h s <' h t
      h< (η< p) = f< p
      h< (midr {x} {y} p) = proj₁ (proj₂ (<'-dense (h< p)))
      h< (midl p) = proj₂ (proj₂ (<'-dense (h< p)))
```

A.2 Axiomatisations

We give Agda implementations of the axiomatisations of inductive-inductive, inductive-recursive and inductive-inductive-recursive definitions respectively, together with some example codes.

We will use the following options:

```
{-# OPTIONS --without-K #-}
{-# OPTIONS --no-positivity-check #-}
{-# OPTIONS --sized-types #-}
```

We use `--without-K` to show that we can, and hence that we should be compatible with homotopy Type Theory. The option `--no-positivity-check` is needed since Agda is not clever enough to see that `ArgA` and `ArgB` only use their arguments in strictly positive position. We use sized types (i.e. `--sized-types`) to convince Agda that `repAbar` below is terminating. At some points, we will also turn off the termination checker, since Agda cannot see that recursive calls are only done at smaller arguments.

A.2.1 Prelude

We start by introducing some basic definitions. Most of these can be found in the standard library 0.7, sometimes with less preferable names chosen, sometimes without universe polymorphism. For these reasons, we prefer to define our own versions.

```
open import Function
open import Relation.Binary.PropositionalEquality hiding ([_])
open Relation.Binary.PropositionalEquality.≡-Reasoning
open import Data.Product
open import Level renaming (zero to zeroL ; suc to sucL)
open import Size

record ⊤ {a : Level} : Set a where

data _+_ (A B : Set) : Set where
```

```
  inl : A -> A + B
  inr : B -> A + B

infixr 40 _+_

[_,_] : ∀ {a} → {A B : Set}{C : A + B -> Set a} ->
                 ((a : A) -> C (inl a)) ->
                 ((b : B) -> C (inr b))
                   -> (c : (A + B)) -> C c
[ f , g ] (inl a) = f a
[ f , g ] (inr b) = g b

data ⊥ : Set where

⊥-elim : ∀ {a} → {A : ⊥ -> Set a} -> (x : ⊥) -> A x
⊥-elim ()

data N₂ : Set where
  tt : N₂
  ff : N₂

J : {a b : Level}{A : Set a} -> (P : (x y : A) -> x ≡ y -> Set b) ->
    (x : A) -> P x x refl -> (y : A) -> (p : x ≡ y) -> P x y p
J P x px .x refl = px


cong₂' : {a b c : Level} {A : Set a} {B : A -> Set b} {C : Set c}
         (f : (x : A) → B x → C) {x y : A} {u : B x}{v : B y} →
         (p : x ≡ y) → subst B p u ≡ v → f x u ≡ f y v
cong₂' f refl refl = refl
```

A.2.2 Inductive-inductive definitions

```
module indind where

  -----------------------------------------------------------------------
  data SPA (Xref : Set) : Set1 where
    nilA : SPA Xref
    nonind : (K : Set) -> (γ : K -> SPA Xref) -> SPA Xref
    A-ind : (K : Set) ->  (γ : SPA (Xref + K)) -> SPA Xref
    B-ind : (K : Set)  -> (h : K -> Xref) -> (γ : SPA Xref) -> SPA Xref

  SPA' : Set1
  SPA' = SPA ⊥

  ArgA : (Xref : Set) ->
         (γA : SPA Xref) ->
         (X : Set)(Y : X -> Set) ->
         (repX : Xref -> X) -> Set
```

```
ArgA Xref nilA X Y repX = ⊤
ArgA Xref (nonind K γ) X Y repX = Σ[ e ∈ K ] ArgA Xref (γ e) X Y repX
ArgA Xref (A-ind K γ) X Y repX
   = Σ[ j ∈ (K -> X) ] ArgA (Xref + K) γ X Y [ repX , j ]
ArgA Xref (B-ind K h γ) X Y repX
   = Σ[ j ∈ ((e : K) -> Y (repX (h e))) ] ArgA Xref γ X Y repX

ArgA' : (γA : SPA') -> (X : Set) -> (Y : X -> Set) -> Set
ArgA' γA X Y = ArgA ⊥ γA X Y ⊥-elim



-- morphism part of the functor ArgA
ArgAfun : {Xref : Set} ->
       (γ : SPA Xref) ->
       {X : Set}{Y : X -> Set}
       {repX : Xref -> X} ->
       {X* : Set}{Y* : X* -> Set} ->
       {repX* : Xref -> X*} ->
       (g : X -> X*)(g' : (a : X) -> Y a -> Y* (g a)) ->
       (p : (e : Xref) -> Y* (g (repX e)) -> Y* (repX* e)) ->
       ArgA Xref γ X Y repX  -> ArgA Xref γ X* Y* repX*
ArgAfun nilA g g' p _ = _
ArgAfun (nonind K γ) g g' p (k , y)
  = (k , ArgAfun (γ k) g g' p y)
ArgAfun (A-ind K γ) g g' p (j , y)
  = (g ∘ j , ArgAfun γ g g' ([ p , (λ k -> id) ]) y)
ArgAfun (B-ind K h γ) {repX = repX} g g' p (j , y)
  = ((λ k -> p (h k) (g' (repX (h k)) (j k))) , ArgAfun γ g g' p y)


ArgAfun' : (γA : SPA') ->
           {X : Set}{Y : X -> Set} ->
           {X* : Set}{Y* : X* -> Set} ->
           (f : X -> X*)(g : (a : X) -> Y a -> Y* (f a)) ->
           ArgA' γA X Y -> ArgA' γA X* Y*
ArgAfun' γA f g = ArgAfun γA f g ⊥-elim


-----------------------------------------------------------


mutual
  data Aterm (γ : SPA' ) (Xref : Set) (Yref : Set) : {i : Size} -> Set where
    aref : ∀ {i} -> Xref -> Aterm γ Xref Yref {↑ i}
    bref : ∀ {i} -> Yref -> Aterm γ Xref Yref {↑ i}
    arg  : ∀ {i} -> ArgA' γ (Aterm γ Xref Yref {i})
                            (Bterm γ {Xref} {Yref})
                                       -> Aterm γ Xref Yref {↑ i}
  Bterm : ∀ {i} -> (γ : SPA') -> {Xref Yref : Set} ->
          Aterm γ Xref Yref {i} -> Set
  Bterm γ (aref y) = ⊥
  Bterm γ (bref y) = ⊤
  Bterm γ (arg y) = ⊥

mutual
  repAbar : ∀ {i} ->  (γ : SPA') ->
            {Xref : Set} -> {Yref : Set} ->
            {X : Set} -> {Y : X -> Set} ->
            (introA : ArgA' γ X Y -> X) ->
            (repX : Xref -> X) ->
            (repindex : Yref -> X) ->
            (repY  : (b : Yref) -> Y (repindex b)) ->
            Aterm γ Xref Yref {i} -> X
  repAbar γ introA repX repindex repY (aref y) = repX y
  repAbar γ introA repX repindex repY (bref y) = repindex y
  repAbar γ introA repX repindex repY (arg {i} y)
                   = introA
                    (ArgAfun' γ
                      (repAbar {i} γ introA repX repindex repY)
                      (repBbar γ introA repX repindex repY)
                      y)

  repBbar : ∀ {i} ->  (γ : SPA') ->
            {Xref : Set} -> {Yref : Set} ->
            {X : Set} -> {Y : X -> Set} ->
            (introA : ArgA' γ X Y -> X) ->
            (repX : Xref -> X) ->
            (repindex : Yref -> X) ->
            (repY  : (b : Yref) -> Y (repindex b)) ->
            (t : Aterm γ Xref Yref {i})
               -> Bterm γ t -> Y (repAbar γ introA repX repindex repY t)
  repBbar γ introA repX repindex repY (aref a) = ⊥-elim
  repBbar γ introA repX repindex repY (bref b) = λ _ -> repY b
  repBbar γ introA repX repindex repY (arg y)  = ⊥-elim


data SPB (Xref : Set)(Yref : Set)(γA : SPA') : Set1 where
  nilB :  Aterm γA Xref Yref -> SPB Xref Yref γA
  nonind : (K : Set) -> (γ : K -> SPB Xref Yref γA) -> SPB Xref Yref γA
  A-ind : (K : Set) ->  (γ : SPB (Xref + K) Yref γA) -> SPB Xref Yref γA
  B-ind : (K : Set) -> (h : (k : K) -> Aterm γA Xref Yref) ->
          (γ : SPB Xref (Yref + K) γA) -> SPB Xref Yref γA

SPB' : (γA : SPA') -> Set1
SPB' = SPB ⊥ ⊥

ArgB : (γA : SPA') ->
```

```
       (Xref : Set)(Yref : Set) ->
       (γB : SPB Xref Yref γA) ->
       (X : Set)(Y : X -> Set)(inA : ArgA' γA X Y -> X)
       (repX : Xref -> X) ->
       (repIndex : Yref -> X) ->
       (repY : (x : Yref) -> Y (repIndex x))
         -> Set
ArgB γA Xref Yref (nilB a) X Y inA repX repIndex repY  = ⊤
ArgB γA Xref Yref (nonind K γ) X Y inA repX repIndex repY
  = Σ[ e ∈ K ] ArgB γA Xref Yref (γ e) X Y inA repX repIndex repY
ArgB γA Xref Yref (A-ind K γ) X Y inA repX repIndex repY
  = Σ[ j ∈ (K -> X) ]
       ArgB γA (Xref + K) Yref γ X Y inA [ repX , j ] repIndex repY
ArgB γA Xref Yref (B-ind K h γ) X Y inA repX repIndex repY
  = Σ[ j ∈ ((e : K) -> Y (repAbar γA inA repX repIndex repY (h e))) ]
       ArgB γA Xref (Yref + K) γ X Y inA repX
                   [ repIndex , ((repAbar γA inA repX repIndex repY) ∘ h) ]
                   [ repY , j ]

ArgB' : (γA : SPA') ->
        (γB : SPB' γA) ->
        (X : Set)(Y : X -> Set)(inA : ArgA' γA X Y -> X) -> Set
ArgB' γA γB X Y inA = ArgB γA ⊥ ⊥ γB X Y inA ⊥-elim ⊥-elim ⊥-elim


Index : {γA : SPA'} ->
        {Xref : Set}{Yref : Set} ->
        (γB : SPB Xref Yref γA) ->
        {X : Set}{Y : X -> Set}{inA : ArgA' γA X Y -> X} ->
        {repX : Xref -> X} ->
        {repIndex : Yref -> X} ->
        {repY : (x : Yref) -> Y (repIndex x)} ->
        ArgB γA Xref Yref γB X Y inA repX repIndex repY -> X
Index {γA = γA} (nilB a) {inA = inA} {repX} {repIndex} {repY} _
  = repAbar γA inA repX repIndex repY a
Index (nonind K γ)(k , y) = Index (γ k) y
Index (A-ind K γ) (j , y) = Index γ y
Index (B-ind K h γ) (j , y) = Index γ y


---------------------------------------------------------------------
-- Introduction rules
---------------------------------------------------------------------
mutual

  data A (γA : SPA')(γB : SPB' γA) : Set where
    introA : ArgA' γA (A γA γB) (B γA γB) -> (A γA γB)

  data B (γA : SPA')(γB : SPB' γA) : (A γA γB) -> Set where
```

```
      introB : (b : ArgB' γA γB (A γA γB) (B γA γB) introA)
                                                      -> (B γA γB) (Index γB b)
```

A.2.2.1 Examples

```
module examples-indind where

  open indind

  -- Encoding multiple constructors into one
  _++_ : {Aref : Set} -> SPA Aref -> SPA Aref -> SPA Aref
  γ ++ ψ = nonind N₂ (λ { tt → γ ; ff → ψ })

  _+++_ : {Aref Bref : Set}{γA : SPA'} ->
          SPB Aref Bref γA -> SPB Aref Bref γA -> SPB Aref Bref γA
  γ +++ ψ = nonind N₂ (λ { tt → γ ; ff → ψ })

  infixr 40 _++_
  infixr 40 _+++_

  -- Single inductive arguments
  A-ind1 : {Aref : Set} -> SPA (Aref + ⊤) -> SPA Aref
  A-ind1 γ = A-ind ⊤ γ

  A-indB1 : {γA : SPA'}{Aref Bref : Set} -> SPB (Aref + ⊤) Bref γA
              -> SPB Aref Bref γA
  A-indB1 γ = A-ind ⊤ γ

  B-ind1 : {Aref : Set} -> Aref -> SPA Aref -> SPA Aref
  B-ind1 i γ = B-ind ⊤ (λ _ → i) γ

  B-indB1 : {γA : SPA'}{Aref Bref : Set} -> Aterm γA Aref Bref
              -> SPB Aref (Bref + ⊤) γA -> SPB Aref Bref γA
  B-indB1 i γ = B-ind ⊤ (λ _ → i) γ

  -- Non-dependent non-inductive arguments
  nonind' : {Aref : Set} -> (K : Set) -> (γ : SPA Aref) -> SPA Aref
  nonind' K γ = nonind K (λ _ → γ)

  nonindB' : ∀ {Aref Bref γA} -> (K : Set) -> (γ : SPB Aref Bref γA)
                -> SPB Aref   Bref γA
  nonindB' K γ = nonind K (λ _ → γ)

  ---------------------------------------------------------------------
  -- Examples
  ---------------------------------------------------------------------

  ---------------------Ctxt and Types------------------
  γCtxt : SPA'
```

```
γCtxt = nilA ++ A-ind1 (B-ind1 (inr _) nilA)

γTy : SPB' γCtxt
γTy =     A-indB1 (nilB  (aref (inr _)))
      +++ A-indB1
            (B-indB1 (aref (inr _))
                     (B-indB1
                       (arg  (ff , ((λ _ → bref (inr _)) , ((λ _ → _) , _))))
                       (nilB  (aref (inr _)))))

Ctxt : Set
Ctxt = A γCtxt γTy

Ty : Ctxt -> Set
Ty = B γCtxt γTy


ε : Ctxt
ε = introA (tt , _)

cons : (Γ : Ctxt) -> Ty Γ -> Ctxt
cons Γ σ = introA ((ff , (λ _ → Γ) , (λ _ → σ) , _))

ι : {Γ : Ctxt} -> Ty Γ
ι {Γ} = introB (tt , ((λ _ → Γ) , _))

Π : (Γ : Ctxt) -> (A : Ty Γ) -> (B : Ty (cons Γ A)) -> Ty Γ
Π Γ A B = introB (ff , ((λ _ → Γ) , ((λ _ → A) , ((λ _ → B) , _))))

---------------------Natural numbers----------------
γNat : SPA'
γNat = nilA ++ A-ind ⊤ nilA

γDummy : SPB' γNat
γDummy = A-indB1 (nilB (aref (inr _)))


ℕ : Set
ℕ = A γNat γDummy

zero : ℕ
zero = introA (tt , _)

suc : ℕ -> ℕ
suc n = introA (ff , ((λ _ → n) , _))

---------------------Finite sets---------------------

γN' : SPA'
```

```
  γℕ' = nonind ℕ (λ n → nilA)

  γFin : SPB' γℕ'
  γFin =     nonind ℕ (λ n → nilB (arg (suc n , _)))
         +++ nonind ℕ (λ n → B-indB1 (arg (n , _)) (nilB (arg ((suc n) , _))))

  ℕ' : Set
  ℕ' = A γℕ' γFin

  i : ℕ -> ℕ'
  i n = introA (n , _)

  Fin : ℕ -> Set
  Fin n = B γℕ' γFin (i n)

  fz : (n : ℕ) -> Fin (suc n)
  fz n = introB (tt , (n , _))

  fsuc : (n : ℕ) -> Fin n -> Fin (suc n)
  fsuc n m = introB (ff , n , ((λ _ → m) , _))
```

A.2.2.2 Simple elimination rules

```
module elim-indind-simple where

  open indind

  IHA : {Aref : Set}(γA : SPA Aref) ->
        {A : Set}{B : A -> Set} ->
        {repA : Aref → A} ->
        (P : A -> Set)(Q : (a : A) -> B a -> Set) ->
        ArgA Aref γA A B repA -> Set
  IHA nilA P Q _ = ⊤
  IHA (nonind K γ) P Q (k , y) = IHA (γ k) P Q y
  IHA (A-ind K γ) P Q (j , y) = ((k : K) -> P (j k)) × IHA γ P Q y
  IHA (B-ind K h γ) {repA = repA} P Q (j , y)
    = ((k : K) -> Q (repA (h k)) (j k)) × IHA γ P Q y

  IHA' : (γA : SPA') ->
         {A : Set}{B : A -> Set} ->
         (P : A -> Set)(Q : (a : A) -> B a -> Set) ->
         ArgA' γA A B -> Set
  IHA' γA = IHA γA {repA = ⊥-elim}

  IHB :  (γA : SPA'){Aref Bref : Set} -> (γB : SPB Aref Bref γA) ->
        {A : Set}{B : A -> Set}{inA : ArgA' γA A B -> A} ->
        {repA : Aref → A} ->
        {repIndex : Bref -> A} ->
        {repB : (x : Bref) -> B (repIndex x)} ->
```

```
      (P : A -> Set)(Q : (a : A) -> B a -> Set) ->
      ArgB γA Aref Bref γB A B inA repA repIndex repB -> Set
IHB γA (nilB _) P Q _ = ⊤
IHB γA (nonind K γ) P Q (k , y) = IHB γA (γ k) P Q y
IHB γA (A-ind K γ) P Q (j , y) = ((k : K) -> P (j k)) × IHB γA γ P Q y
IHB γA (B-ind K h γ) {inA = inA} {repA} {repIndex} {repB} P Q (j , y)
  = ((k : K) -> Q (repAbar γA inA repA repIndex repB (h k)) (j k))
       × IHB γA γ P Q y

IHB' : (γA : SPA')(γB : SPB' γA) ->
      {A : Set}{B : A -> Set}{inA : ArgA' γA A B -> A} ->
      (P : A -> Set)(Q : (a : A) -> B a -> Set) ->
      ArgB' γA γB A B inA -> Set
IHB' γA γB P Q y = IHB γA γB P Q y


mapIHA :  {Aref : Set}(γA : SPA Aref) ->
          {A : Set}{B : A -> Set} ->
          {repA : Aref -> A} ->
          {P : A -> Set}{Q : (a : A) -> B a -> Set} ->
          (f : (a : A) -> P a)(g : (a : A) -> (b : B a) -> Q a b) ->
          (a : ArgA Aref γA A B repA) -> IHA γA P Q a
mapIHA nilA f g _ = _
mapIHA (nonind K γ) f g (k , y) = mapIHA (γ k) f g y
mapIHA (A-ind K γ) f g (j , y) = (f ∘ j , mapIHA γ f g y)
mapIHA (B-ind K h γ) {repA = repA} f g (j , y)
  = ((λ k → g (repA (h k)) (j k)) , mapIHA γ f g y)

mapIHA' : (γA : SPA') ->
          {A : Set}{B : A -> Set} ->
          {P : A -> Set}{Q : (a : A) -> B a -> Set} ->
          (f : (a : A) -> P a)(g : (a : A) -> (b : B a) -> Q a b) ->
          (a : ArgA' γA A B) -> IHA' γA P Q a
mapIHA' γA = mapIHA γA


mapIHB : (γA : SPA'){Aref Bref : Set} -> (γB : SPB Aref Bref γA) ->
         {A : Set}{B : A -> Set}{inA : ArgA' γA A B -> A} ->
         {repA : Aref → A} ->
         {repIndex : Bref -> A} ->
         {repB : (x : Bref) -> B (repIndex x)} ->
         {P : A -> Set}{Q : (a : A) -> B a -> Set} ->
         (f : (a : A) -> P a)(g : (a : A) -> (b : B a) -> Q a b) ->
         (y : ArgB γA Aref Bref γB A B inA repA repIndex repB)
            -> IHB γA γB P Q y
mapIHB γA (nilB _) f g _ = _
mapIHB γA (nonind K γ) f g (k , y) = mapIHB γA (γ k) f g y
mapIHB γA (A-ind K γ)  f g (j , y) = (f ∘ j , mapIHB γA γ f g y)
mapIHB γA (B-ind K h γ) {inA = inA} {repA} {repIndex} {repB} f g (j , y)
```

```
    = ((λ k → g (repAbar γA inA repA repIndex repB (h k)) (j k)) ,
       mapIHB γA γ f g y)

  mapIHB' : (γA : SPA' )(γB : SPB' γA) ->
            {A : Set}{B : A → Set}{inA : ArgA' γA A B -> A} ->
            {P : A -> Set} ->
            {Q : (a : A) -> B a -> Set} ->
            (f : (a : A) -> P a)(g : (a : A) -> (b : B a) -> Q a b) ->
            (y : ArgB' γA γB A B inA) -> IHB' γA γB P Q  y
  mapIHB' γA γB f g y  = mapIHB γA γB f g y

  {-# NO_TERMINATION_CHECK #-}
  mutual
     elimA : (γA : SPA')(γB : SPB' γA) ->
             (P : (A γA γB) -> Set) ->
             (Q : (a : (A γA γB)) -> (b : (B γA γB) a) -> Set) ->
             (stepA : (x : ArgA' γA (A γA γB) (B γA γB))
                                   -> IHA' γA P Q x -> P (introA x)) ->
             (stepB : (y : ArgB' γA γB (A γA γB) (B γA γB) introA) ->
                    (ybar : IHB' γA γB P Q y) -> Q (Index γB y) (introB y)) ->
             (a : (A γA γB)) -> P a
     elimA γA γB P Q stepA stepB (introA a)
       = stepA a (mapIHA' γA  (elimA γA γB P Q stepA stepB)
                              (elimB γA γB P Q stepA stepB) a)

     elimB : (γA : SPA')(γB : SPB' γA) ->
             (P : (A γA γB) -> Set) ->
             (Q : (a : (A γA γB)) -> (b : (B γA γB) a) -> Set) ->
             (stepA : (x : ArgA' γA (A γA γB) (B γA γB))
                                   -> IHA' γA P Q x -> P (introA x)) ->
             (stepB : (y : ArgB' γA γB (A γA γB) (B γA γB) introA) ->
                      (ybar : IHB' γA γB P Q y) -> Q (Index γB y) (introB y)) ->
             (a : A γA γB) -> (b : B γA γB a) ->  Q a b
     elimB γA γB P Q stepA stepB .(Index γB b) (introB b)
       = stepB b (mapIHB' γA γB (elimA γA γB P Q stepA stepB)
                                (elimB γA γB P Q stepA stepB) b)
```

A.2.3 Inductive-recursive definitions

```
module IR (D : Set1) where

  data OP : Set1 where
    ι : D -> OP
    σ : (A : Set) -> (f : A -> OP) -> OP
    δ : (A : Set) -> (F : (A -> D) -> OP) -> OP

  ⟦_⟧₀ : OP -> (U : Set)(T : U -> D) -> Set
  ⟦_⟧₀ (ι d) U T = ⊤
  ⟦_⟧₀ (σ A f) U T = Σ[ a ∈ A ] ⟦ f a ⟧₀ U T
  ⟦_⟧₀ (δ A F) U T = Σ[ g ∈ (A -> U) ] ⟦ F (T ∘ g) ⟧₀ U T

  ⟦_⟧₁ : (γ : OP) -> (U : Set)(T : U -> D) -> ⟦ γ ⟧₀ U T -> D
  ⟦_⟧₁ (ι d) U T _ = d
  ⟦_⟧₁ (σ A f) U T (a , x) = ⟦ f a ⟧₁ U T x
  ⟦_⟧₁ (δ A F) U T (g , x) = ⟦ F (T ∘ g) ⟧₁ U T x
```

A.2.4 Inductive-inductive-recursive definitions

```
-------------------------------------------
open import Relation.Binary.PropositionalEquality.TrustMe

ext : {a b : Level} -> Extensionality a b
ext p = trustMe
-------------------------------------------


module IIR (D : Set1) where

  -----------------------------------------------------------------------
  data SPA (Xref : Set) : Set1 where
    nilA : D -> SPA Xref
    nonind : (K : Set) -> (γ : K -> SPA Xref) -> SPA Xref
    A-ind : (K : Set) ->  (γ : (K -> D) -> SPA (Xref + K)) -> SPA Xref
    B-ind : (K : Set)  -> (h : K -> Xref) -> (γ : SPA Xref) -> SPA Xref

  SPA' : Set1
  SPA' = SPA ⊥

  ArgA : (Xref : Set) ->
         (γA : SPA Xref) ->
         (X : Set)(Y : X -> Set) ->
         (T : X -> D) ->
         (repX : Xref -> X) -> Set
  ArgA Xref (nilA _) X Y T repX = ⊤
  ArgA Xref (nonind K γ) X Y T repX = Σ[ e ∈ K ] ArgA Xref (γ e) X Y T repX
  ArgA Xref (A-ind K γ) X Y T repX
    = Σ[ j ∈ (K -> X) ] ArgA (Xref + K) (γ (T ∘ j)) X Y T [ repX , j ]
  ArgA Xref (B-ind K h γ) X Y T repX
    = Σ[ j ∈ ((e : K) → Y (repX (h e))) ] ArgA Xref γ X Y T repX

  ArgA' : (γA : SPA') -> (X : Set) -> (Y : X -> Set) -> (T : X -> D) -> Set
  ArgA' γA X Y T = ArgA ⊥ γA X Y T ⊥-elim


  {- morphism part of the functor ArgA -}
  ArgAfun : {Xref : Set} ->
         (γ : SPA Xref) ->
         {X : Set}{Y : X -> Set}{T : X -> D}
         {repX : Xref -> X} ->
```

```
       {X* : Set}{Y* : X* -> Set}{T* : X* -> D} ->
       {repX* : Xref -> X*} ->
       (f : X -> X*)(g : (a : X) -> Y a -> Y* (f a)) ->
       (coh : T ≡ T* ∘ f) ->
       (p : (e : Xref) -> Y* (f (repX e)) -> Y* (repX* e)) ->
       ArgA Xref γ X Y T repX  -> ArgA Xref γ X* Y* T* repX*
ArgAfun (nilA _) f g coh p _ = _
ArgAfun (nonind K γ) f g coh p (k , y)
  = (k , ArgAfun (γ k) f g coh p y)
ArgAfun {Xref} (A-ind K γ) {T = T}  {repX* = repX*} f g coh p (j , y)
  = (f ∘ j , subst (λ z -> ArgA (Xref + K) z _ _ _ [ repX* , f ∘ j ])
                   (cong (λ w -> γ (w ∘ j)) coh)
                   (ArgAfun (γ (T ∘ j)) f g coh [ p , (λ k -> id) ] y))
ArgAfun {Xref} (B-ind K h γ) {repX = repX} f g coh p (j , y)
  = ((λ k -> p (h k) (g (repX (h k)) (j k))) , ArgAfun γ f g coh p y)

ArgAfun' : (γA : SPA') ->
        {X : Set}{Y : X -> Set}{T : X -> D} ->
        {X* : Set}{Y* : X* -> Set}{T* : X* -> D} ->
        (f : X -> X*)(g : (a : X) -> Y a -> Y* (f a)) ->
        (coh : T ≡ T* ∘ f) ->
        ArgA' γA X Y T -> ArgA' γA X* Y* T*
ArgAfun' γA f g coh = ArgAfun γA f g coh ⊥-elim

{- "recursive part" -}
FunA : {Xref : Set} ->
       (γA : SPA Xref) ->
       {X : Set}{Y : X -> Set} ->
       {T : X -> D} ->
       {repX : Xref -> X} ->
       ArgA Xref γA X Y T repX -> D
FunA (nilA o)  _ = o
FunA (nonind K γ) (k , x) = FunA (γ k) x
FunA (A-ind K γ) {T = T} (j , x) = FunA (γ (T ∘ j)) x
FunA (B-ind K h γ) (j , x) = FunA γ x

FunA' : (γA : SPA') -> (X : Set) -> (Y : X -> Set) -> (T : X -> D) ->
        ArgA' γA X Y T -> D
FunA' γA X Y T = FunA γA {X} {Y} {T}

FunA-coh : {Xref : Set} ->
           (γA : SPA Xref) ->
           {X : Set}{Y : X -> Set}{T : X -> D}
           {repX : Xref -> X} ->
           {X* : Set}{Y* : X* -> Set}{T* : X* -> D} ->
           {repX* : Xref -> X*} ->
           {f : X -> X*}{g : (a : X) -> Y a -> Y* (f a)} ->
           {coh : T ≡ T* ∘ f} ->
           {p : (e : Xref) -> Y* (f (repX e)) -> Y* (repX* e)} ->
           (x : ArgA Xref γA X Y T repX) ->
             FunA γA x ≡ FunA γA (ArgAfun γA {T* = T*} f g coh p x)
FunA-coh (nilA o) x = refl
FunA-coh (nonind K γ) (k , x) = FunA-coh (γ k) x
FunA-coh {Xref} (A-ind K γ) {T = T} {repX = repX} {Y* = Y*} {T*}
         {repX* = repX*} {f = f} {g = g} {coh = coh} {p = p} (j , x)
  = begin
      FunA (A-ind K γ) (j , x)
    ≡⟨ refl ⟩
      FunA (γ  (T ∘ j)) x
    ≡⟨ FunA-coh (γ (T ∘ j)) x ⟩
      FunA (γ  (T ∘ j)) (ArgAfun (γ (T ∘ j)) f g coh [ p , (λ k -> id) ] x)
    ≡⟨ cong₂' (λ a b -> FunA a b) (cong (λ w -> γ (w ∘ j)) coh) refl ⟩
      FunA (γ  (T* ∘ f ∘ j))
           (subst (λ z -> ArgA (Xref + K) z _ _ _ [ repX* , (f ∘ j) ])
                  (cong (λ w -> γ (w ∘ j)) coh)
                  (ArgAfun (γ (T ∘ j)) f g coh [ p , (λ k -> id) ] x))
    ≡⟨ refl ⟩
      FunA (A-ind K γ)
           (ArgAfun (A-ind K γ) {Y* = Y*} {T* = T*}  f g coh p (j , x))
    ∎
FunA-coh (B-ind K h γ) (j , x) = FunA-coh γ x


------------------------------------------------------------


mutual
  data Aterm (γ : SPA' ) (Xref : Set)(Yref : Set)
             (TrefA : Xref -> D)(TrefB : Yref -> D) : {i : Size} -> Set where
    aref : ∀ {i} -> Xref -> Aterm γ Xref Yref TrefA TrefB {↑ i}
    bref : ∀ {i} -> Yref -> Aterm γ Xref Yref TrefA TrefB {↑ i}
    arg  : ∀ {i} -> ArgA' γ (Aterm γ Xref Yref TrefA TrefB {i})
                            (Bterm γ) (Tterm γ)
                                       -> Aterm γ Xref Yref TrefA TrefB {↑ i}

  Bterm : ∀ {i} -> (γ : SPA') ->
          {Xref Yref : Set} ->
          {TrefA : Xref -> D} -> {TrefB : Yref -> D} ->
          Aterm γ Xref Yref TrefA TrefB {i} -> Set
  Bterm γ (aref y) = ⊥
  Bterm γ (bref y) = ⊤
  Bterm γ (arg y) = ⊥


  Tterm : ∀ {i} -> (γ : SPA') ->
          {Xref Yref : Set} ->
          {TrefA : Xref -> D} -> {TrefB : Yref -> D} ->
          Aterm γ Xref Yref TrefA TrefB {i} -> D
  Tterm γA {TrefA = TrefA} (aref y) = TrefA y
  Tterm γA {TrefB = TrefB} (bref y) = TrefB y
  Tterm γA (arg x) = FunA' γA _ _ _ x



mutual
  repAbar : ∀ {i} ->(γ : SPA') ->
            {Xref : Set}{Yref : Set} ->
            {TrefA : Xref -> D}{TrefB : Yref -> D} ->
            {X : Set}{Y : X -> Set}{T : X -> D} ->
            (introA : ArgA' γ X Y T -> X) ->
            (repX : Xref -> X) ->
            (repindex : Yref -> X) ->
            (repY  : (b : Yref) -> Y (repindex b)) ->
            {T-sane : (x : ArgA' γ X Y T) -> T (introA x) ≡ FunA γ x} ->
            {TrefA-sane : (x : Xref) -> TrefA x ≡ T (repX x)} ->
            {TrefB-sane : (x : Yref) -> TrefB x ≡ T (repindex x)} ->
            Aterm γ Xref Yref TrefA TrefB {i}
           -> X
  repAbar γ introA repX repindex repY (aref y) = repX y
  repAbar γ introA repX repindex repY (bref y) = repindex y
  repAbar γ {T = T} introA repX repindex repY {T-sane} {TrefA-sane} {TrefB-sane} (arg {i} y)
                   = introA
                    (ArgAfun' γ
                      (repAbar {i} γ introA repX repindex repY)
                      (repBbar γ introA repX repindex repY)
                      (ext (coh γ {T = T} T-sane TrefA-sane TrefB-sane))
                      y)


  repBbar : ∀ {i} ->(γ : SPA') ->
            {Xref : Set}{Yref : Set} ->
            {TrefA : Xref -> D}{TrefB : Yref -> D} ->
            {X : Set}{Y : X -> Set}{T : X -> D} ->
            (introA : ArgA' γ X Y T -> X) ->
            (repX : Xref -> X) ->
            (repindex : Yref -> X) ->
            (repY  : (b : Yref) -> Y (repindex b)) ->
            {T-sane : (x : ArgA' γ X Y T) -> T (introA x) ≡ FunA γ x} ->
            {TrefA-sane : (x : Xref) -> TrefA x ≡ T (repX x)} ->
            {TrefB-sane : (x : Yref) -> TrefB x ≡ T (repindex x)} ->
            (t : Aterm γ Xref Yref TrefA TrefB {i}) ->
              Bterm γ t -> Y (repAbar γ introA repX repindex repY
                                        {T-sane} {TrefA-sane} {TrefB-sane} t)
  repBbar γ introA repX repindex repY (aref a) = ⊥-elim
  repBbar γ introA repX repindex repY (bref b) = λ _ -> repY b
  repBbar γ introA repX repindex repY (arg y)  = ⊥-elim
  coh : ∀ {i} ->(γ : SPA') ->
        {Xref : Set}{Yref : Set} ->
        {TrefA : Xref -> D}{TrefB : Yref -> D} ->
        {X : Set}{Y : X -> Set}{T : X -> D} ->
        {introA : ArgA' γ X Y T -> X} ->
        {repX : Xref -> X} ->
        {repindex : Yref -> X} ->
        {repY  : (b : Yref) -> Y (repindex b)} ->
        (T-sane : (x : ArgA' γ X Y T) -> T (introA x) ≡ FunA γ x) ->
        (TrefA-sane : (x : Xref) -> TrefA x ≡ T (repX x)) ->
        (TrefB-sane : (x : Yref) -> TrefB x ≡ T (repindex x)) ->
        (x : Aterm γ Xref Yref TrefA TrefB {i}) ->
        Tterm γ x ≡ T (repAbar γ introA repX repindex repY
                                 {T-sane} {TrefA-sane} {TrefB-sane} x)
  coh γ _ TrefA-sane _ (aref x) = TrefA-sane x
  coh γ _ _ TrefB-sane (bref x) = TrefB-sane x
  coh γ {T = T} {introA} {repX} {repIndex} {repY} T-sane _ _(arg w)
     = begin
         Tterm γ (arg w)
       ≡⟨ refl ⟩
         FunA γ w
       ≡⟨ FunA-coh γ w ⟩
         FunA γ (ArgAfun' γ _ _ _ w)
       ≡⟨ sym (T-sane (ArgAfun' γ _ _ _ w)) ⟩
         T (introA (ArgAfun' γ _ _ _ w))
       ≡⟨ refl ⟩
         T (repAbar γ introA repX repIndex repY (arg w))
       ■
```

```
data SPB (Xref : Set)(Yref : Set)
         (TrefA : Xref -> D)(TrefB : Yref -> D)(γA : SPA') : Set1 where
  nilB :  Aterm γA Xref Yref TrefA TrefB -> SPB Xref Yref TrefA TrefB γA
  nonind : (K : Set) -> (γ : K -> SPB Xref Yref TrefA TrefB γA)
                                              -> SPB Xref Yref TrefA TrefB γA
  A-ind : (K : Set) ->
          (γ : (t : K -> D) -> SPB (Xref + K) Yref [ TrefA , t ] TrefB γA)
                                              -> SPB Xref Yref TrefA TrefB γA
  B-ind : (K : Set) -> (h : (k : K) -> Aterm γA Xref Yref TrefA TrefB) ->
          (γ : (t : K -> D) -> SPB Xref (Yref + K) TrefA [ TrefB , t ] γA)
                                              -> SPB Xref Yref TrefA TrefB γA

SPB' : (γA : SPA') -> Set1
SPB' = SPB ⊥ ⊥ ⊥-elim ⊥-elim

ArgB : (γA : SPA') ->
       (Xref : Set)(Yref : Set) ->
       {TrefA : Xref -> D}{TrefB : Yref -> D} ->
       (X : Set)(Y : X -> Set)(T : X -> D)(inA : ArgA' γA X Y T -> X) ->
       (repX : Xref -> X) ->
       (repIndex : Yref -> X) ->
       (repY : (x : Yref) -> Y (repIndex x)) ->
       (γB : SPB Xref Yref TrefA TrefB γA) ->
       (T-sane : (x : ArgA' γA X Y T) -> T (inA x) ≡ FunA γA x) ->
       (TrefA-sane : (x : Xref) -> TrefA x ≡ T (repX x)) ->
       (TrefB-sane : (x : Yref) -> TrefB x ≡ T (repIndex x))
        -> Set
ArgB γA Xref Yref X Y T inA repX repIndex repY (nilB a)
                            T-sane TrefA-sane TrefB-sane = ⊤
ArgB γA Xref Yref X Y T inA repX repIndex repY (nonind K γ)
                            T-sane TrefA-sane  TrefB-sane
  = Σ[ e ∈ K ] ArgB γA Xref Yref X Y T inA repX repIndex repY (γ e)
                                           T-sane TrefA-sane TrefB-sane
ArgB γA Xref Yref X Y T inA repX repIndex repY (A-ind K γ)
                            T-sane TrefA-sane  TrefB-sane
  = Σ[ j ∈ (K -> X) ]
     ArgB γA (Xref + K) Yref X Y T inA [ repX , j ] repIndex repY (γ (T ∘ j ))
                             T-sane [ TrefA-sane , (λ k -> refl) ]  TrefB-sane
ArgB γA Xref Yref X Y T inA repX repIndex repY (B-ind K h γ)
                            T-sane TrefA-sane  TrefB-sane
  = Σ[ j ∈ ((e : K) -> Y (repAbar γA inA repX repIndex repY (h e))) ]
      ArgB γA Xref (Yref + K) X Y T inA repX
             [ repIndex , ((repAbar γA inA repX repIndex repY {T-sane}) ∘ h) ]
             [ repY , j ]
             (γ (T ∘ (repAbar γA inA repX repIndex repY
                                 {T-sane} {TrefA-sane} {TrefB-sane}) ∘ h))
             T-sane TrefA-sane
             [ TrefB-sane , (λ k -> refl) ]




ArgB' : (γA : SPA') ->
        (γB : SPB' γA) ->
        (X : Set)(Y : X -> Set)(T : X -> D)(inA : ArgA' γA X Y T -> X) ->
        (T-sane : (x : ArgA' γA X Y T) -> T (inA x) ≡ FunA γA x) -> Set
ArgB' γA γB X Y T inA T-sane
  = ArgB γA ⊥ ⊥ X Y T inA ⊥-elim ⊥-elim ⊥-elim γB T-sane ⊥-elim ⊥-elim


Index : {γA : SPA'} ->
        {Xref : Set}{Yref : Set}
        {TrefA : Xref -> D}{TrefB : Yref -> D} ->
        {X : Set}{Y : X -> Set}{T : X -> D}{inA : ArgA' γA X Y T -> X} ->
        {repX : Xref -> X} ->
        {repIndex : Yref -> X} ->
        {repY : (x : Yref) -> Y (repIndex x)} ->
```

```
          (γB : SPB Xref Yref TrefA TrefB γA) ->
          {T-sane : (x : ArgA' γA X Y T) -> T (inA x) ≡ FunA γA x} ->
          {TrefA-sane : (x : Xref) -> TrefA x ≡ T (repX x)} ->
          {TrefB-sane : (x : Yref) -> TrefB x ≡ T (repIndex x)} ->
          ArgB γA Xref Yref X Y T inA repX repIndex repY γB
                                      T-sane TrefA-sane TrefB-sane -> X
  Index {γA} {inA = inA} {repX} {repIndex} {repY} (nilB a)
                         {T-sane = T-sane} {TrefA-sane} {TrefB-sane} _
    = repAbar γA inA repX repIndex repY
                     {T-sane = T-sane} {TrefA-sane} {TrefB-sane} a
  Index (nonind K γ)(k , y) = Index (γ k) y
  Index {T = T} (A-ind K γ) (j , y) = Index (γ (T ∘ j)) y
  Index {γA} {T = T} {inA} {repX} {repIndex} {repY} (B-ind K h γ) (j , y)
    = Index (γ (T ∘ (repAbar γA inA repX repIndex repY) ∘ h)) y


  mutual

    data A (γA : SPA')(γB : SPB' γA) : Set where
      introA : ArgA' γA (A γA γB) (B γA γB) (T γA γB) -> (A γA γB)

    {-# NO_TERMINATION_CHECK #-}
    T : (γA : SPA')(γB : SPB' γA) -> A γA γB -> D
    T γA γB (introA x) = FunA' γA (A γA γB) (B γA γB) (T γA γB) x

    data B (γA : SPA')(γB : SPB' γA) : (A γA γB) -> Set where
      introB : (b : ArgB' γA γB (A γA γB) (B γA γB) (T γA γB)
                              introA (λ x -> refl))
                                           -> (B γA γB) (Index γB b)
```

A.2.4.1 Examples

```
  _++_ : {Xref : Set} -> SPA Xref -> SPA Xref -> SPA Xref
  γ ++ ψ = nonind N₂ (λ { tt → γ ; ff → ψ })


  _+++_ : ∀ {Xref Yref TrefA TrefB γA} -> SPB Xref Yref TrefA TrefB γA -> SPB Xref Yref Tre
  γ +++ ψ = nonind N₂ (λ { tt → γ ; ff →  ψ })

  infixr 40 _++_
  infixr 40 _+++_

module examples-indind-as-IIR where

  open IIR ⊤

  ---------------------Ctxt and Types------------------
  γCtxt : SPA'
```

```
γCtxt = nilA _ ++ A-ind ⊤ (λ _ → B-ind ⊤ (λ _ → inr _) (nilA _))

γTy : SPB' γCtxt
γTy =     A-ind ⊤ (λ _ → nilB  (aref (inr _)))
      +++ A-ind ⊤
           (λ _ → B-ind
                    ⊤
             (λ _ → aref (inr _))
             (λ _ → B-ind
                      ⊤
               (λ _ → arg  (ff , ((λ _ → bref (inr _)) , ((λ _ → _) , _))))
               (λ _ → nilB  (aref (inr _)))))


Ctxt : Set
Ctxt = A γCtxt γTy

Ty : Ctxt -> Set
Ty = B γCtxt γTy


ε : Ctxt
ε = introA (tt , _)

cons : (Γ : Ctxt) -> Ty Γ -> Ctxt
cons Γ σ = introA ((ff , (λ _ → Γ) , (λ _ → σ) , _))

ι : {Γ : Ctxt} -> Ty Γ
ι {Γ} = introB (tt , (λ _ → Γ) , _)

Π : (Γ : Ctxt) -> (A : Ty Γ) -> (B : Ty (cons Γ A)) -> Ty Γ
Π Γ A B = introB (ff , ((λ _ → Γ ) , ((λ _ → A) , ((λ _ → B) , _))))


---------------------Natural numbers----------------
γNat : SPA'
γNat = nilA _ ++ A-ind ⊤ (λ _ → nilA _)

γDummy : SPB' γNat
γDummy = A-ind ⊤ (λ _ → nilB (aref (inr _)))

ℕ : Set
ℕ = A γNat γDummy

Nzero : ℕ
Nzero = introA (tt , _)

Nsuc : ℕ -> ℕ
Nsuc n = introA (ff , ((λ _ → n) , _))
```

```
  --------------------Finite sets--------------------

  γℕ' : SPA'
  γℕ' = nonind ℕ (λ n → nilA _)

  γFin : SPB' γℕ'
  γFin =     nonind ℕ (λ n → nilB (arg (Nsuc n , _)))
         +++ nonind ℕ (λ n → B-ind ⊤ (λ _ → arg (n , _))
                                      (λ _ → nilB (arg ((Nsuc n) , _))))

  ℕ' : Set
  ℕ' = A γℕ' γFin

  i : ℕ -> ℕ'
  i n = introA (n , _)

  Fin : ℕ -> Set
  Fin n = B γℕ' γFin (i n)

  fz : (n : ℕ) -> Fin (Nsuc n)
  fz n = introB (tt , (n , _))

  fsuc : (n : ℕ) -> Fin n -> Fin (Nsuc n)
  fsuc n m = introB (ff , n , ((λ _ → m) , _))

module examples-indrec-as-IIR where

  open IIR Set

  γℕΣ : SPA'
  γℕΣ =    nilA examples-indind-as-IIR.ℕ
        ++ A-ind ⊤ (λ X → A-ind (X _) (λ Y → nilA (Σ (X _) Y)))

  γDummy : SPB' γℕΣ
  γDummy = A-ind ⊤ (λ _ → nilB (aref (inr _)))

  UℕΣ : Set
  UℕΣ = A γℕΣ γDummy

  TℕΣ : UℕΣ -> Set
  TℕΣ = T γℕΣ γDummy

  n : UℕΣ
  n = introA (tt , _)

  sigma : (a : UℕΣ) -> (b : TℕΣ a -> UℕΣ) -> UℕΣ
  sigma a b = introA (ff , (λ _ → a) , b , _)
  private
    T-n : TℕΣ n ≡ examples-indind-as-IIR.ℕ
    T-n = refl

    T-sigma : ∀ {a b} -> TℕΣ (sigma a b) ≡ Σ (TℕΣ a) (TℕΣ ∘ b)
    T-sigma = refl
```

A.2.4.2 Embedding inductive-inductive and inductive-recursive definitions

```
module IItoIIR where

  open module indindrec =  IIR ⊤

  open indind

  ΨA : {Xref : Set} -> indind.SPA Xref -> indindrec.SPA Xref
  ΨA nilA = nilA _
  ΨA (nonind K γ) = nonind K (λ x → ΨA (γ x))
  ΨA (A-ind K γ) = A-ind K (λ _ → ΨA γ)
  ΨA (B-ind K h γ) = B-ind K h (ΨA γ)

  ΨA-correct : {Aref : Set} ->
               (γA : indind.SPA Aref) ->
               {A : Set}{B : A → Set} ->
               {repA : Aref → A} ->
               indind.ArgA Aref γA A B repA
                                      ≡ indindrec.ArgA Aref (ΨA γA) A B _ repA
  ΨA-correct nilA = refl
  ΨA-correct (nonind K γ) = cong (λ z → Σ _ z) (ext (λ k → ΨA-correct (γ k)))
  ΨA-correct (A-ind K γ) = cong (λ z → Σ _ z) (ext (λ j → ΨA-correct γ))
  ΨA-correct (B-ind K h γ) = cong (λ z → Σ _ z) (ext (λ j → ΨA-correct γ))

  ΨArgA : (γA : indind.SPA') ->
          {A : Set}{B : A → Set} ->
          {A* : Set}{B* : A* → Set} ->
          (f : A -> A*)(g : (x : A) -> B x -> B* (f x)) ->
          indind.ArgA' γA A B -> indindrec.ArgA' (ΨA γA) A* B* _
  ΨArgA γA f g = indindrec.ArgAfun' (ΨA γA) f g refl
                                                  ∘ (subst id (ΨA-correct γA))

  ΨArgA-inv : (γA : indind.SPA') ->
              {A : Set}{B : A → Set} ->
              indindrec.ArgA' (ΨA γA) A B _ -> indind.ArgA' γA A B
  ΨArgA-inv γA x = (subst id (sym (ΨA-correct γA)) x)


  mutual
    ΨATerm : ∀ {Xref Yref γA i} ->
             indind.Aterm γA Xref Yref {i}
```

```
                -> indindrec.Aterm (ΨA γA) Xref Yref _ _ {i}
    ΨATerm (aref x) = aref x
    ΨATerm (bref x) = bref x
    ΨATerm {γA = γA} (arg x) = arg (ΨArgA γA ΨATerm ΨBTerm x)

    ΨBTerm : ∀ {Xref Yref γA i} -> (x : indind.Aterm γA Xref Yref {i}) ->
             indind.Bterm γA x -> indindrec.Bterm (ΨA γA) (ΨATerm x)
    ΨBTerm (aref x) y = y
    ΨBTerm (bref x) y = y
    ΨBTerm (arg x) y = y


  ΨB : {Xref Yref : Set}{γA : indind.SPA'} ->
       indind.SPB Xref Yref γA -> indindrec.SPB Xref Yref _ _ (ΨA γA)
  ΨB (nilB a) = nilB (ΨATerm a)
  ΨB (nonind K γ) = nonind K (λ z → ΨB (γ z))
  ΨB (A-ind K γ) = A-ind K (λ _ → ΨB γ)
  ΨB (B-ind K h γ) = B-ind K (ΨATerm ∘ h) (λ _ → ΨB γ)

module IRtoIIR (D : Set1) where

  open IIR D

  open IR D

  Φ : ∀ {Xref} -> OP -> SPA Xref
  Φ (ι d) = nilA d
  Φ (σ A f) = nonind A (λ z → Φ (f z))
  Φ (δ A F) = A-ind A (λ z → Φ (F z))

  Φ-correctU : {U : Set}{B : U -> Set}{T : U -> D} ->
               {Xref : Set}{repA : Xref -> U} ->
               (γ : OP) -> ⟦ γ ⟧0 U T ≡ ArgA Xref (Φ γ) U B T repA
  Φ-correctU (ι d) = refl
  Φ-correctU (σ A f) = cong (λ z → Σ A z) (ext (λ a → Φ-correctU (f a)))
  Φ-correctU {U = U} {T = T} (δ A F) = cong (λ z → Σ (A → U) z)
                                            (ext (λ g → Φ-correctU (F (T ∘ g))))

  ΦU : ∀ {Xref U T B repA} -> (γ : OP) -> ⟦ γ ⟧0 U T
            -> ArgA Xref (Φ γ) U B T repA
  ΦU (ι d) _ = _
  ΦU (σ A f) (a , x) = (a , ΦU (f a) x)
  ΦU {T = T} (δ A F) (g , x) = (g , ΦU (F (T ∘ g)) x)

  Φ-correctT : {U : Set}{B : U -> Set}{T : U -> D} ->
               {Xref : Set}{repX : Xref -> U} ->
               (γ : OP) -> (x : ⟦ γ ⟧0 U T) ->
               ⟦ γ ⟧1 U T x ≡ FunA (Φ γ) {Y = B} {repX = repX} (ΦU γ x)
  Φ-correctT (ι d) _ = refl
```

```
Φ-correctT (σ A f) (a , x) = Φ-correctT (f a) x
Φ-correctT {T = T} (δ A F) (g , x) = Φ-correctT (F (T ∘ g)) x
```

Bibliography

Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003. (cited on p. 97).

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, pages 23–38, 2003. (cited on pp. 97, 104).

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In *Automata, Languages and Programming, 31st International Colloqium (ICALP)*, pages 59 – 71, 2004. (cited on pp. 5, 97).

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3 – 27, 2005. (cited on pp. 5, 97).

Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002. (cited on p. 156).

Peter Aczel. An introduction to inductive definitions. In *Handbook of Mathematical Logic*, pages 739–782. Elsevier, 1977. (cited on pp. 8, 96).

Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77*. North-Holland, 1978. (cited on p. 151).

Peter Aczel. Frege structures and the notions of proposition, truth and set. In Jon Barwise, H. Jerome Keisler, and Kenneth Kunen, editors, *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 31 – 59. Elsevier, 1980. (cited on p. 5).

Peter Aczel. On relating type theories and set theories. *Lecture Notes In Computer Science*, 1657:1–18, 1999. (cited on p. 89).

Peter Aczel and Michael Rathjen. *Notes on Constructive Set Theory*. Draft, 2010. (cited on p. 117).

Jiří Adámek. Free algebras and automata realizations in the language of categories. *Comment. Math. Univ. Carolinae*, 15(1074):589–602, 1974. (cited on p. 91).

Jiri Adámek, Stefan Milius, and Lawrence Moss. Initial algebras and terminal coalgebras: a survey. Draft, June 2010. (cited on pp. 92, 136).

Thorsten Altenkirch. Extensional equality in intensional type theory. In *Logic in Computer Science*, pages 412 – 420, 1999. (cited on p. 28).

Thorsten Altenkirch and Peter Morris. Indexed containers. In *Logic In Computer Science*, pages 277 –285, 2009. (cited on pp. 5, 97).

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57 – 68. ACM, 2007. (cited on p. 25).

Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In Andrea Corradini, Bartek Klin, and Corina Cirstea, editors, *Conference on Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 70 – 84. Springer, 2011. (cited on pp. 11, 59).

Roland Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In A. Avron, R. Harper, F. Honsell, I. Mason, and G. Plotkin, editors, *Proceedings of the Workshop on general logic, Edinburgh, February, 1987*, volume ECS-LFCS-88-52, 1988. (cited on p. 4).

Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989. (cited on p. 4).

Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. (cited on p. 15).

Jean Bénabou. Fibered categories and the foundations of naive category theory. *The Journal of Symbolic Logic*, 50(1):10 – 37, 1985. (cited on p. 77).

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265–269, 2003. (cited on p. 5).

Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill, 1967. (cited on p. 3).

Errett Bishop and Douglas Bridges. *Constructive analysis*. Berlin, 1985. (cited on p. 153).

Rodney Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969. (cited on p. 4).

Venanzio Capretta. Wander types : A formalization of coinduction-recursion. *Progress in Informatics*, (10):47–64, 2013. (cited on p. 173).

Aurelio Carboni and Peter Johnstone. Connected limits, familial representability and Artin glueing. *Mathematical Structures in Computer Science*, 5(04):441 – 459, 1995. (cited on p. 161).

John Cartmell. *Generalised algebraic theories and contextual categories*. PhD thesis, Oxford University, 1978. (cited on p. 66).

James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. (cited on pp. 31, 136).

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *ICFP*, volume 45, pages 3–14. ACM, 2010. (cited on pp. 5, 173).

Pierre Clairambault. From categories with families to locally cartesian closed categories. Technical report, ENS Lyon, 2006. (cited on p. 71).

Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. In *TLCA*, 2011. (cited on pp. 71, 72, 74, 76, 77).

John Conway. *On numbers and games*. AK Peters, 2001. (cited on pp. 147, 149, 156, 157, 159).

Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 85–92, 1992. (cited on pp. 29, 156).

Thierry Coquand and Huet Gérard. The calculus of constructions. *Information and Computation*, 76:95 – 120, 1988. (cited on p. 5).

Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer Berlin Heidelberg, 1990. (cited on p. 5).

Haskell Curry. Functionality in combinatory logic. In *Proceedings of the National Academy of Sciences*, pages 584 – 590, 1934. (cited on p. 22).

Haskell Curry and Robert Feys. *Combinatory Logic Vol. I*. Amsterdam: North-Holland, 1958. (cited on p. 22).

Pierre-Evariste Dagand and Conor McBride. Elaborating inductive definitions. In *Journées Francophones des Langages Applicatifs*, 2013. (cited on pp. 29, 58).

Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. *Lecture Notes in Computer Science*, 4502:93–109, 2007. (cited on pp. 31, 119, 120, 136, 138).

Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381 – 392, 1972. (cited on p. 14).

Nicolaas Govert de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189 – 204, 1991. (cited on p. 14).

Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280 – 206. Cambridge University Press, 1991. (cited on p. 5).

Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994. (cited on p. 5).

Peter Dybjer. Internal type theory. *Lecture Notes in Computer Science*, 1158:120–134, 1996. (cited on pp. 66, 70, 71).

Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1-2):329–335, 1997. (cited on pp. 5, 97, 100).

Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000. (cited on pp. 5, 32).

Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed lambda calculi and applications: 4th international conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999: proceedings*, pages 129–146. Springer Verlag, 1999. (cited on pp. 5, 35, 36, 89, 94, 119, 131, 132, 133).

Peter Dybjer and Anton Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1-3):1–47, 2003. (cited on pp. 5, 39, 60, 69, 87).

Peter Dybjer and Anton Setzer. Indexed induction–recursion. *Journal of logic and algebraic programming*, 66(1):1–49, 2006. (cited on pp. 5, 38, 89, 144).

Linus Ek, Ola Holmström, and Stevan Andjelkovic. Formalizing Arne Andersson trees and left-leaning Red-Black trees in Agda. Bachelor thesis, Chalmers Institute of Technology, 2009. (cited on p. 38).

Clement Fumex. *Induction and Coinduction schemes in Category Theory*. PhD thesis, University of Strathclyde, 2012. (cited on p. 87).

Richard Garner. On the strength of dependent products in the type theory of Martin-Löf. *Annals of Pure and Applied Logic*, 160(1):1–12, 2009. (cited on p. 20).

Herman Geuvers. Induction is not derivable in second order dependent type theory. In *Typed lambda calculi and applications: 5th international conference, TLCA 2001, Kraków, Poland, May 2-5, 2001: proceedings*, pages 166–181. Springer, 2001. (cited on p. 5).

Neil Ghani and Peter Hancock. Containers, monads and induction recursion. To appear in MSCS, 2012. (cited on p. 151).

Neil Ghani, Patricia Johann, and Clement Fumex. Fibrational induction rules for initial algebras. In *Computer Science Logic*, pages 336–350. Springer, 2010. (cited on p. 87).

Neil Ghani, Patricia Johann, and Clement Fumex. Indexed induction and coinduction, fibrationally. In Andrea Corradini, Bartek Klin, and Corina Cirstea, editors, *Conference on Algebras and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2011. (cited on p. 87).

Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg. Positive inductive-recursive definitions. In Reiko Heckel and Stefan Milius, editors, *CALCO 2013*, volume 8089 of *Lecture Notes in Computer Science*, pages 19 – 33, 2013a. (cited on pp. 12, 147).

Neil Ghani, Lorenzo Malatesta, Fredrik Nordvall Forsberg, and Anton Setzer. Fibred data types. In *Logic in Computer Science*, pages 243 – 252, 2013b. (cited on pp. 8, 173).

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989. ISBN 0-521-37181-3. (cited on p. 23).

Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994. (cited on p. 21).

Healfdene Goguen, Conor McBride, and James McKinna. *Eliminating dependent pattern matching*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006. (cited on pp. 30, 157).

Joseph A. Goguen, James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977. ISSN 0004-5411. (cited on pp. 39, 59).

Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987. (cited on p. 61).

Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion, indexed containers and dependent polynomials are equivalent. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications*, volume 7941 of *Lecture Notes in Computer Science*, pages 156 – 172. Springer, 2013. (cited on pp. 162, 163).

Ronald Harrop. Concerning formulas of the types $A \to B \vee C$, $A \to (Ex)B(x)$ in intuitionistic formal systems. *The Journal of Symbolic Logic*, 25(1):27 – 32, 1960. (cited on p. 28).

Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145(2):107 – 152, 1998. (cited on pp. 69, 73, 87).

Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *Computer Science Logic*, pages 427–441. Springer, 1994. (cited on p. 77).

Martin Hofmann. Syntax and semantics of dependent types. In Andrew Pitts and Peter Dybjer, editors, *Semantics and Logics of Computation*, pages 79 – 130. Cambridge University Press, 1997. (cited on pp. 66, 74).

Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In G. Sambin and J. Smith, editors, *Twenty five years of constructive type theory*, pages 83–111, Oxford, 1998. Oxford University Press. (cited on p. 24).

William Howard. The formulae-as-types notion of construction. Published in in Seldin, J., Hindley, R.: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1969. (cited on p. 22).

Gérard Huet and Amokrane Saïbi. Constructive category theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg*, 1998. (cited on p. 30).

IBM Applied Science Division. Specifications for the IBM mathematical formula translating system, FORTRAN. Technical report, International Business Machines Corporation, 1954. (cited on p. 2).

Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169 – 207, 1993. (cited on p. 66).

Bart Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Elsevier, 1999. (cited on p. 87).

Reinhard Kahle and Anton Setzer. An extended predicative definition of the Mahlo universe. In Ralf Schindler, editor, *Ways of Proof Theory. Festschrift on the occasion of Wolfram Pohler's retirement*, Ontos Series in Mathematical Logic. Ontos Verlag, 2010. (cited on p. 173).

Gregory Maxwell Kelly. Elementary observations on 2-categorical limits. *Bulletin of the Australian Mathematical Society*, 39(02):301–317, 1989. (cited on p. 61).

Stephen Cole Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. Wolters-Noordhoff, 1952. (cited on p. 9).

Donald Erwin Knuth. *Surreal Numbers*. Addison Wesley, 1974. (cited on p. 147).

Joachim Lambek. Subequalizers. *Canadian Mathematical Bulletin*, 13:337 – 349, 1970. (cited on p. 61).

Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University press, 1994. (cited on p. 21).

Zhaohui Luo. Notes on universes in type theory. Lecture notes for a talk at Institute for Advanced Study, Princeton (URL: `http://www.cs.rhul.ac.uk/home/zhaohui/universes.pdf`), 2012. (cited on p. 16).

Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: theory and implementation. *Information and Computation*, 223:18 – 42, 2012. (cited on p. 16).

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1998. (cited on pp. 30, 79).

José Pedro Magalhães. *Less Is More: Generic Programming Theory and Practice*. PhD thesis, Universiteit Utrecht, 2012. (cited on p. 5).

Lorenzo Malatesta. *Investigations into Induction-Recursion*. PhD thesis, University of Strathclyde, 2013. Forthcoming. (cited on p. 147).

Lionel Mamane. Surreal numbers in Coq. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs: International Workshop TYPES 2004*, volume 3839 of *Lecture Notes in Computer Science*, pages 170 – 185. Springer, 2006. (cited on pp. 148, 153, 159).

Simon Marlow. Haskell 2010 language report, 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`. (cited on p. 2).

Per Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In Jan Erik Fenstad, editor, *Proceedings of the 2nd Scandinavian logic symposium*, pages 179 – 216, 1971. (cited on p. 4).

Per Martin-Löf. An intuitionistic theory of types. Published in Twenty-Five Years of Constructive Type Theory, 1972. (cited on pp. 2, 3, 4, 5, 22, 38).

Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104:153–175, 1982. (cited on pp. 2, 3, 4).

Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis Naples, 1984. (cited on pp. 2, 4, 16).

Nax Paul Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1987. (cited on p. 5).

Kenji Miyamoto, Fredrik Nordvall Forsberg, and Helmut Schwichtenberg. Program extraction from nested definitions. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 370 – 385. Springer, 2013. (cited on p. 172).

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007. (cited on p. 5).

Peter Morris, Thorsten Altenkirch, and Neil Ghani. A universe of strictly positive families. *International Journal of Foundations of Computer Science*, 20(1):83–107, 2009. (cited on p. 172).

Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990. (cited on pp. 2, 13, 25).

Bengt Nordström, Kent Petersson, and Jan Smith. Martin-Löf's type theory. In *Handbook of Logic in Computer Science: Logic and algebraic methods*. Oxford University Press, 2001. (cited on p. 2).

Fredrik Nordvall Forsberg and Anton Setzer. Inductive-inductive definitions. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 2010. (cited on pp. 11, 31, 35, 89).

Fredrik Nordvall Forsberg and Anton Setzer. A finite axiomatisation of inductive-inductive definitions. In Ulrich Berger, Hannes Diener, Peter Schuster, and Monika Seisenberger, editors, *Logic, Construction, Computation*, volume 3 of *Ontos mathematical logic*, pages 259 – 287. Ontos Verlag, 2012. (cited on pp. 12, 31, 35, 89, 148).

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007. (cited on p. 28).

Erik Palmgren. On universes in type theory. In Giovanni Sambin and Jan Smith, editors, *Twenty five years of constructive type theory*, pages 191 – 204. Oxford University Press, 1998. (cited on p. 38).

Erik Palmgren. Constructivist and structuralist foundations: Bishop's and Lawvere's theories of sets. *Annals of Pure and Applied Logic*, 163(10):1384 – 1399, 2012. (cited on p. 30).

Alan Perlis and Klaus Samelson. Preliminary report: International Algebraic Language. *Communications of the ACM*, 1(12):8 – 22, 1958. (cited on p. 2).

Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics 1989*, pages 209–228. Springer-Verlag, 1990. (cited on p. 5).

D. Prawitz. Proofs and the meaning and completeness of the logical constants. In Jaakko Hintikka, Ilkka Niiniluoto, and Esa Saarinen, editors, *Essays on Mathematical and Philosophical Logic*, pages 25–40. Reidel, 1979. (cited on p. 4).

Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study*. Almquist & Wiksell, 1965. (cited on p. 13).

Frank Rosemeier. On Conway numbers and generalized real numbers. In Ulrich Berger, Horst Oswald, and Peter Schuster, editors, *Reuniting the Antipodes*, pages 211 – 227. Kluwer Academic Publishers, 2001. (cited on pp. 148, 153, 160).

Bertrand Russel. *The Principles of Mathematics*. Cambridge University Press, 1903. (cited on p. 2).

Anne Salvesen and Jan Smith. The strength of the subset type in Martin-Löf's type theory. In *Logic in Computer Science*, 1988. (cited on p. 28).

Dana Scott. Constructive validity. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275. Springer, 1970. (cited on p. 22).

Anton Setzer. An upper bound for the proof theoretical strength of Martin-Löf type theory with W-type and one universe. Draft, 1996. (cited on p. 110).

Anton Setzer. Well-ordering proofs for Martin-Löf's type theory with W-type and one universe. *Annals of Pure and Applied Logic*, 92:113 – 159, 1998. (cited on p. 38).

Anton Setzer. Universes in type theory part I – Inaccessibles and Mahlo. In Alessandro Andretta, Keith Kearnes, and Domenico Zambella, editors, *Logic Colloquium '04*, number 29 in Lecture Notes in Logic, pages 123 – 156. Cambridge University Press, 2008. (cited on pp. 38, 173).

Paul Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, 1999. (cited on p. 66).

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013. (cited on pp. 25, 27, 148, 160).

Anne Sjerp Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Number 344 in Lecture Notes in Mathematics. Springer, 1973. (cited on p. 28).

Anne Sjerp Troelstra. Definability of finite sum types in Martin-Löf's type theories. *Indagationes Mathematicae*, 86(4):475 – 481, 1983. (cited on p. 21).

Anne Sjerp Troelstra. On the syntax of Martin-Löf's type theories. *Theoretical Computer Science*, 51:1 – 26, 1987. (cited on p. 14).

Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in mathematics: an introduction*, volume 121 of *Studies in logic and the foundations of mathematics*. Elsevier Science, 1988. (cited on p. 22).

Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, L'Universite Paris 7, 1994. (cited on p. 21).

Olov Wilander. Constructing a small category of setoids. *Mathematical Structures in Computer Science*, 22:103 – 121, 2012. (cited on p. 30).

Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009. (cited on p. 20).

Index