A Categorical Semantics for Inductive-Inductive Definitions

Thorsten Altenkirch[1⋆,⋆⋆], Peter Morris[1⋆⋆], Fredrik Nordvall Forsberg[2⋆], and Anton Setzer[2⋆]

[1] School of Computer Science, University of Nottingham, UK
[2] Department of Computer Science, Swansea University, UK

Abstract. Induction-induction is a principle for defining data types in Martin-Löf Type Theory. An inductive-inductive definition consists of a set A, together with an A-indexed family $B : A \to \mathrm{Set}$, where both A and B are inductively defined in such a way that the constructors for A can refer to B and vice versa. In addition, the constructors for B can refer to the constructors for A. We extend the usual initial algebra semantics for ordinary inductive data types to the inductive-inductive setting by considering dialgebras instead of ordinary algebras. This gives a new and compact formalisation of inductive-inductive definitions, which we prove is equivalent to the usual formulation with elimination rules.

1 Introduction

Induction is an important principle of definition and reasoning, especially so in constructive mathematics and computer science, where the concept of inductively defined set and data type coincide. There are two well-established approaches to model the semantics of such data types: in Martin-Löf Type Theory [14], each set A comes equipped with an eliminator which at the same time represents reasoning by induction over A and the definition of recursive functions out of A. A more categorical approach [10] models data types as initial T-algebras for a suitable endofunctor T.

At first, it would seem that the eliminator approach is stronger, as it allows us to define dependent functions $(x : A) \to P(x)$, in contrast with the non-dependent arrows $A \to B$ given by the initiality of the algebra. However, Hermida and Jacobs [12] showed that an eliminator can be defined for every initial T-algebra, where T is a polynomial functor. Ghani et. al. [9] then extended this to arbitrary endofunctors. This covers many forms of induction and data type definitions such as indexed inductive definitions [5] and induction-recursion [7] (Dybjer and Setzer [8] also give a direct proof for induction-recursion).

There are, however, other meaningful forms of data types which are not covered by these results. One such example are inductive-inductive definitions [16], where a set A and a function $B : A \to \mathrm{Set}$ are simultaneously inductively defined (compare

⋆ Supported by EPSRC grant EP/G033374/1.
⋆⋆ Supported by EPSRC grant EP/G034109/1.

with induction-recursion, where A is defined inductively and B recursively). In addition, the constructors for B can refer to the constructors for A.

In earlier work [16], a subset of the authors gave an eliminator-based axiomatisation of a type theory with inductive-inductive definitions and showed it to be consistent. In this article, we describe a generalised initial algebra semantics for induction-induction, and prove that it is equivalent to the original axiomatisation.

One could imagine that that inductive-inductive definitions could be described by functors mapping families of sets to families of sets (similar to the situation for induction-recursion [8]), but this fails to take into account that the constructors for B should be able to refer to the constructors for A. Thus, we will see that the constructor for B can be described by an operation

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

where $c : \mathrm{Arg_A}(A, B) \to A$ refers to the already defined constructor for A. However, $(\mathrm{Arg_A}, \mathrm{Arg_B})$ is then no longer an endofunctor and we move to the more general setting of dialgebras [11, 18] to describe algebras of such functors. The equivalence between initiality and having an eliminator still carries over to this new setting.

1.1 Examples of Inductive-Inductive Definitions

Danielsson [4] and Chapman [3] define 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. Let us concentrate on contexts and types for simplicity. There should be an empty context ε, and if we have any context Γ and a valid type σ in that context, then we should be able to extend the context with a fresh variable of that type. We end up with the following inductive definition of the set of contexts:

$$\frac{}{\varepsilon : \mathrm{Ctxt}} \qquad \frac{\Gamma : \mathrm{Ctxt} \quad \sigma : \mathrm{Type}(\Gamma)}{\Gamma \rhd \sigma : \mathrm{Ctxt}}$$

For types, let us have a base type ι (valid in any context) and a dependent function type: if σ is a type in context Γ, and τ is a type in Γ extended with a fresh variable of type σ (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 $\mathrm{Type} : \mathrm{Ctxt} \to \mathrm{Set}$:

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

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

For an example with more of a programming flavour, consider defining 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

$\leqslant_{\mathrm{L}}$: $(\mathbb{N} \times \mathrm{SortedList}) \to \mathrm{Set}$ with $n \leqslant_{\mathrm{L}} \ell$ 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 $\mathrm{cons}(n, \ell, p)$. Translated into introduction rules, this becomes:

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

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

$$\frac{}{\mathrm{triv}_m : m \leqslant_{\mathrm{L}} \mathrm{nil}} \qquad \frac{q : m \leqslant n \qquad p_{m,\ell} : m \leqslant_{\mathrm{L}} \ell}{\ll q, p_{m,\ell} \gg_{m,n,\ell,p} : m \leqslant_{\mathrm{L}} \mathrm{cons}(n, \ell, p)}$$

Of course, there are many alternative ways to define such a data type using ordinary induction, but the inductive-inductive one seems natural and might be more convenient for some purposes. It is certainly more pleasant to work with in the proof assistant/ programming language Agda [17] which allows inductive-inductive definitions using the `mutual` keyword. One aim of our investigation into inductive-inductive definitions is to justify their existence in Agda.

It might be worth pointing out that inductive-inductive and inductive-recursive definitions are different. Not every inductive-inductive definition can be directly translated into an inductive-recursive definition, since the inductive definition of the second type B may not proceed according to the recursive ordering. The contexts and types example above is an example of this. On the other hand, inductive-recursive definitions can use negative occurrences of B, which is not possible for inductive-inductive definitions. For instance, a universe closed under Π-types can be defined using induction-recursion but not induction-induction.

1.2 Preliminaries and notation

We work in an extensional type theory [15] with the following ingredients:

Set We use Set to denote our universe of small types, and we write $B : A \to \mathrm{Set}$ for an A-indexed family of sets.

***Π*-types** Given $A : \mathrm{Set}$ and $B : A \to \mathrm{Set}$, then $\big((x : A) \to B(x)\big) : \mathrm{Set}$. Elements of $(x : A) \to B(x)$ are functions f that map $a : A$ to $f(a) : B(a)$.

***Σ*-types** Given $A : \mathrm{Set}$ and $B : A \to \mathrm{Set}$, then $\Sigma x : A.\, B(x) : \mathrm{Set}$. Elements of $\Sigma x : A.\, B(x)$ are dependent pairs $\langle a, b \rangle$ where $a : A$ and $b : B(a)$. We write $\pi_0 : \Sigma x : A.\, B(x) \to A$ and $\pi_1 : (y : \Sigma x : A.\, B(x)) \to B(\pi_0(y))$ for the projections. We write $\{\, a : A \mid B(a) \}$ for $\Sigma x : A.\, B(x)$ if $B : A \to \mathrm{Set}$ is propositional, i.e. there is at most one inhabitant in $B(a)$ for every $a : A$.

+ Given $A, B : \mathrm{Set}$, we denote their coproduct $A + B$ with coprojections $\mathrm{inl} : A \to A + B$ and $\mathrm{inr} : B \to A + B$. We use $[f, g]$ for cotupling.

Equality and unit types Given $a, b : A$ we write $a = b : \mathrm{Set}$ for the equality type, inhabitated by refl if and only if $a = b$. In contrast, the unit type $\mathbf{1}$ always has a unique element $\star : \mathbf{1}$.

We call a type expression *strictly positive* in X if X never appears in the domain of a Π-type. It is a requirement for inductive definitions in predicative Type Theory that the inductively defined types appear only strictly positive in the domain of the constructors.

2 Inductive-Inductive Definitions as Dialgebras

In this section, our goal is to describe each inductive-inductively defined set as the initial object in a category constructed from a description of the set. Just as for ordinary induction and initial algebras, this description will be a functor of sorts, but because of the more complicated structure involved, this will no longer be an endofunctor. The interesting complication is the fact that the constructor for the second set B can refer to the constructor for the first set A (as for example the argument $\tau : \mathrm{Type}(\Gamma \rhd \sigma)$ referring to $\cdot \rhd \cdot$ in the introduction rule for the Π-type). Thus we will model the constructor for B as (the second component of) a morphism $(c, d) : \mathrm{Arg}(A, B, c) \to (A, B)$ where $c : \mathrm{Arg}_{\mathrm{A}}(A, B) \to A$ is the constructor for A. Here, (c, d) is a morphism in the category of families of sets:

Definition 2.1. *The category Fam(Set) of families of sets has as objects pairs (A, B), where A is a set and $B : A \to$ Set is an A-indexed family of sets. 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 $U : \mathrm{Fam(Set)} \to \mathrm{Set}$ sending (A, B) to A and (f, g) to f. Now, $c : \mathrm{Arg}_{\mathrm{A}}(A, B) \to A$ is not an $\mathrm{Arg}_{\mathrm{A}}$-algebra, since $\mathrm{Arg}_{\mathrm{A}} : \mathrm{Fam(Set)} \to \mathrm{Set}$ is not an endofunctor. However, we have $c : \mathrm{Arg}_{\mathrm{A}}(A, B) \to U(A, B)$. This means that c is a $(\mathrm{Arg}_{\mathrm{A}}, U)$-dialgebra, as introduced by Hagino [11]:

Definition 2.2. *Let $F, G : \mathbb{C} \to \mathbb{D}$ be functors. The category Dialg(F, G) has as objects pairs (A, f) where $A \in \mathbb{C}$ and $f : F(A) \to G(A)$. A morphism from (A, f) to (A', f') is a morphism $h : A \to A'$ in $\mathbb{C}$ such that $G(h) \circ f = f' \circ F(h)$.*

There is a forgetful functor $V : \mathrm{Dialg}(F, G) \to \mathbb{C}$ defined by $V(A, f) = A$.

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

Definition 2.3. *An inductive-inductive definition is given by two functors*

$$Arg_A : Fam(Set) \to Set \qquad Arg : Dialg(Arg_A, U) \to Fam(Set)$$

such that $U \circ Arg = Arg_A \circ V$.

Since the first functor is determined by the second, we often write such a pair as $\mathrm{Arg} = (\mathrm{Arg}_{\mathrm{A}}, \mathrm{Arg}_{\mathrm{B}})$ where

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

Example 2.4 (Contexts and types). The inductive-inductive definition of Ctxt : Set and $\mathrm{Type} : \mathrm{Ctxt} \to \mathrm{Set}$ from the introduction is given by

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

For $\mathrm{Arg}_{\mathrm{Ctxt}}$, 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 $\Gamma : \mathrm{Ctxt}$ and $\sigma : \mathrm{Type}(\Gamma)$. Similar considerations apply to $\mathrm{Arg}_{\mathrm{Type}}$.

Example 2.5 (Sorted lists). The sorted list example does not fit into our framework, since $\leqslant_{\mathrm{L}}: (\mathbb{N} \times \mathrm{SortedList}) \to \mathrm{Set}$ is indexed by $\mathbb{N} \times \mathrm{SortedList}$ and not simply 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 $B : (\mathbb{N} \times A) \to \mathrm{Set}$. The inductive-inductive definition of $\mathrm{SortedList} : \mathrm{Set}$ and $\leqslant_{\mathrm{L}}: (\mathbb{N} \times \mathrm{SortedList}) \to \mathrm{Set}$ is then given by

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

For ease of presentation, we will only consider ordinary families of sets.

2.1 A Category for Inductive-Inductive Definitions

Given $\mathrm{Arg} = (\mathrm{Arg}_{\mathrm{A}}, \mathrm{Arg}_{\mathrm{B}})$ representing an inductive-inductive definition, we will now construct a category $\mathbb{E}_{\mathrm{Arg}}$ whose initial object (if it exists) is the intended interpretation of the inductive-inductive definition. Figure 1 summarises the functors and categories involved (U, V and W are all forgetful functors).

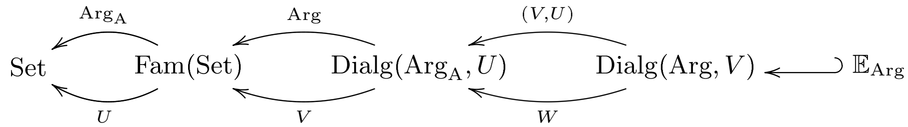

Fig. 1. The functors and categories involved.

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

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

does not look quite right – we need c and d_0 to be the same!

To this end, we will consider the equalizer of the forgetful functor $W : \mathrm{Dialg}(\mathrm{Arg}, V) \to \mathrm{Dialg}(\mathrm{Arg_A}, U)$, $W(A, B, c, (d_0, d_1)) = (A, B, c)$, 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 $U(d_0, d_1) : U(\mathrm{Arg}(A, B, c)) \to U(V(A, B, c))$ but $U \circ \mathrm{Arg} = \mathrm{Arg_A} \circ V$, so that $U(d_0, d_1) : \mathrm{Arg_A}(V(A, B, c)) \to U(V(A, B, c))$. In other words, $(V(A, B, c), U(d_0, d_1))$ is an object in $\mathrm{Dialg}(\mathrm{Arg_A}, U)$, so (V, U) really is a functor from $\mathrm{Dialg}(\mathrm{Arg}, V)$ to $\mathrm{Dialg}(\mathrm{Arg_A}, U)$.

Definition 2.6. *For $Arg = (Arg_A, Arg_B)$ representing an inductive-inductive definition, let $\mathbb{E}_{Arg}$ be the underlying category of the equaliser of (V, U) and the forgetful functor $W : Dialg(Arg, V) \to Dialg(Arg_A, U)$.*

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

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

$$g(c(x), d(x, y)) = d'(\mathrm{Arg_A}(f, g)(x), \mathrm{Arg_B}(f, g)(x, y)) \ .$$

Example 2.7. Consider the functors $\mathrm{Arg_{Ctxt}}$, $\mathrm{Arg_{Type}}$ from Example 2.4:

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

An object in $\mathbb{E}_{(\mathrm{Arg_{Ctxt}},\mathrm{Arg_{Type}})}$ consists of $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$ 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[3]

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

Remark 2.8. The intended interpretation of the inductive-inductive definition given by $\mathrm{Arg} = (\mathrm{Arg_A}, \mathrm{Arg_B})$ is the initial object in $\mathbb{E}_{\mathrm{Arg}}$. Depending on the meta-theory, this might of course not exist. However, we will show that it does if and only if an eliminator for the inductive-inductive definition exists.

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

Remark 2.9. From Figure 1, it should be clear how to generalise the current construction to the simultaneous definition of $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$, $C : (x : A) \to B(x) \to \mathrm{Set}$, etc.: for a definition of n sets, replace Fam(Set) with the category FAM_n of families $(A_1, A_2, A_3, \ldots, A_n)$ and consider $\mathrm{Arg}_{\mathrm{A}} : \mathrm{FAM}_n \to \mathrm{Set}$, $\mathrm{Arg} : \mathrm{Dialg}(\mathrm{Arg}_{\mathrm{A}}, U) \to \mathrm{Fam(Set)}$, $\mathrm{Arg}_{\mathrm{C}} : \mathbb{E}_{\mathrm{Arg}} \to \mathrm{FAM}_3, \ldots$ taking an equalizer where necessary to make the constructors in different positions equal.

2.2 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 the introduction. Suppose that we want to define a concatenation $+\!\!+ : \mathrm{Ctxt} \to \mathrm{Ctxt} \to \mathrm{Ctxt}$ of contexts – such an operation could be useful to formulate more general formation rules, such as:

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

Such an operation should satisfy the equations

$$\begin{aligned} \Delta +\!\!+ \quad \varepsilon \quad &= \Delta \\ \Delta +\!\!+ (\Gamma \rhd \sigma) &= (\Delta +\!\!+ \Gamma) \rhd (\mathrm{wk}_\Gamma(\sigma, \Delta)) \quad , \end{aligned}$$

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

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

Our hope is now to exploit the initiality of (Ctxt, Type) to get such operations. Recall from Example 2.4 that Ctxt, Type are the underlying sets for the inductive-inductive definition given by the functors

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

From the types of $+\!\!+ : \mathrm{Ctxt} \to \mathrm{Ctxt} \to \mathrm{Ctxt}$ and $\mathrm{wk} : (\Gamma : \mathrm{Ctxt}) \to (A : \mathrm{Type}(\Gamma)) \to (\Delta : \mathrm{Ctxt}) \to \mathrm{Type}(\Delta +\!\!+ \Gamma)$, we see that if we can equip (A, B) where $A = \mathrm{Ctxt} \to \mathrm{Ctxt}$ and $B(f) = (\Delta : \mathrm{Ctxt}) \to \mathrm{Type}(f(\Delta))$ with an $(\mathrm{Arg}_{\mathrm{Ctxt}}, \mathrm{Arg}_{\mathrm{Type}})$ 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:

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

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

Since $(A, B, \mathrm{in}_A, \mathrm{in}_B)$ is an object in $\mathbb{E}_{\mathrm{Arg}}$, initiality gives us a morphism $(+\!\!+, \mathrm{wk}) : (\mathrm{Ctxt}, \mathrm{Type}) \to (A, B)$ such that $(+\!\!+, \mathrm{wk}) \circ ([\varepsilon, \rhd], [\iota, \Pi]) = (\mathrm{in}_A, \mathrm{in}_B) \circ (\mathrm{Arg}_{\mathrm{Ctxt}}, \mathrm{Arg}_{\mathrm{Type}})(+\!\!+, \mathrm{wk})$. In particular, this means that

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

Thus, we see that $\Delta +\!\!+ \varepsilon = \Delta$ and $\Delta +\!\!+ (\Gamma \rhd \sigma) = (\Delta +\!\!+ \Gamma) \rhd \mathrm{wk}_\Gamma(\sigma, \Delta)$ as required.[4] In the same way, the equations for the weakening operation hold.

2.3 Relationship to induction-induction as axiomatised in [16]

In short, the earlier axiomatisation [16] postulated the existence of a universes $\mathrm{SP}'_{\mathrm{A}}$, $\mathrm{SP}'_{\mathrm{B}}$ of codes for inductive-inductive sets, together with decoding functions $\mathrm{Arg}'_{\mathrm{A}}$, $\mathrm{Arg}'_{\mathrm{B}}$ and $\mathrm{Index}'_{\mathrm{B}}$. Intuitively, $\mathrm{Arg}'_{\mathrm{A}}$ gives the domain of the constructor $\mathrm{intro}_{\mathrm{A}}$ for A, $\mathrm{Arg}'_{\mathrm{B}}$ the domain for the constructor $\mathrm{intro}_{\mathrm{B}}$ for B and $\mathrm{Index}'_{\mathrm{B}}(x)$ the index of the type of $\mathrm{intro}_{\mathrm{B}}(x)$. More formally, they have types

$$
\begin{aligned}
&\mathrm{Arg}'_{\mathrm{A}} : (\gamma_A : \mathrm{SP}'_{\mathrm{A}})(A : \mathrm{Set})(B : A \to \mathrm{Set}) \to \mathrm{Set} \ , \\
&\mathrm{Arg}'_{\mathrm{B}} : (\gamma_A : \mathrm{SP}'_{\mathrm{A}})(\gamma_B : \mathrm{SP}'_{\mathrm{B}}(\gamma_A)) \\
&\qquad \to (A : \mathrm{Set})(B_0 : A \to \mathrm{Set})(B_1 : \mathrm{Arg}'_{\mathrm{A}}(\gamma_A, A, B_0) \to \mathrm{Set}) \\
&\qquad \to \ldots \to (B_n : \mathrm{Arg}'^{n}_{\mathrm{A}}(\gamma_A, A, \vec{B}_{(n)}) \to \mathrm{Set}) \to \mathrm{Set} \ , \\
&\mathrm{Index}'_{\mathrm{B}}(\gamma_A, \gamma_B, A, B_0, \ldots, B_n) : \\
&\qquad \mathrm{Arg}'_{\mathrm{B}}(\gamma_A, \gamma_B, A, B_0, \ldots, B_n) \to \coprod_{i=0}^{i} \mathrm{Arg}'^{n}_{\mathrm{A}}(\gamma_A, A, \vec{B}_{(i)}) \ ,
\end{aligned}
$$

where $\vec{B}_{(i)} = (B_0, \ldots, B_{i-1})$ and $\mathrm{Arg}'^{i}_{\mathrm{A}}(\gamma_A, A, B_{(i)})$ is defined by

$$
\begin{aligned}
\mathrm{Arg}'^{0}_{\mathrm{A}}(\gamma_A, A, B_{(0)}) &:= A \\
\mathrm{Arg}'^{n+1}_{\mathrm{A}}(\gamma_A, A, \vec{B}_{(n), B_{n+1}}) &:= \mathrm{Arg}'_{\mathrm{A}}(\gamma_A, \coprod_{i=0}^{n} \mathrm{Arg}'^{i}_{\mathrm{A}}(\gamma_A, A, \vec{B}_{(i)}), [B_0, \ldots, B_n]) \ .
\end{aligned}
$$

The axiomatisation then states that we have introduction and elimination rules, i.e. that for each code $\gamma = (\gamma_A, \gamma_B)$ there exists is a family $A_\gamma : \mathrm{Set}$, $B_\gamma : A_\gamma \to \mathrm{Set}$ with constructors $\mathrm{intro}_{\mathrm{A}} : \mathrm{Arg}'_{\mathrm{A}}(\gamma_A, A_\gamma, B_\gamma) \to A_\gamma$ and $\mathrm{intro}_{\mathrm{B}} : (x : \mathrm{Arg}'_{\mathrm{B}}(\gamma, A_\gamma, B_\gamma, B_1, \ldots, B_n)) \to B_\gamma(\mathrm{index}(x))$, and a suitable eliminator (see Section 3). Here, $B_i = B \circ k_i$ and $\mathrm{index}(x) = [k_0, \ldots, k_n](\mathrm{Index}'_{\mathrm{B}}(\gamma, A, B_0, \ldots, B_n, x))$ where $k_0 = \mathrm{id}$ and $k_{i+1} = \mathrm{intro}_{\mathrm{A}} \circ \mathrm{Arg}'^{i}_{\mathrm{A}}([k_0, \ldots, k_i], [\mathrm{id}', \ldots, \mathrm{id}'])$. The codes are chosen so that all occurrences of A and B in the domains of $\mathrm{intro}_{\mathrm{A}}$ and $\mathrm{intro}_{\mathrm{B}}$ are strictly positive.

The relationship between the codes from this axiomatisation and the formalisation in this article can now be summed up in the following proposition:

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

Proposition 2.10. *For each code $\gamma = (\gamma_A, \gamma_B)$, the operations $Arg_{\gamma_A} : Fam(Set) \to Set$ and $Arg_{\gamma} = (Arg_{\gamma_A}, Arg_{\gamma_B}) : Dialg(Arg_{\gamma_A}, U) \to Fam(Set)$ given by*

$$Arg_{\gamma_A}(A, B) := Arg'_A(\gamma_A, A, B) \ ,$$
$$Arg_{\gamma_B}(A, B, c, x) := \{\, y : Arg'_B(\gamma_A, \gamma_B, A, B_0, \ldots, B_n) \mid c(x) = index(y)\}$$

are functorial. □

We will call a functor F strictly positive if it arises as $F = \mathrm{Arg}_\gamma$ for some code γ. In Section 3.3 , we show that that the original introduction and elimination rules hold if and only if $\mathbb{E}_{\mathrm{Arg}_\gamma}$ has an initial object.

3 The Elimination Principle

In this section, we introduce the elimination principle for inductive-inductive definitions. We show that every initial object has an eliminator (Proposition 3.8), and that every object with an eliminator is weakly initial (Proposition 3.9). Under the added assumption of strict positivity, we can also show uniqueness. Hence the two notions are equivalent for strictly positive functors (Theorem 3.10).

3.1 Warm-up: a Generic Eliminator for an Inductive Definition

The traditional type-theoretical way of defining recursive functions like the context concatenation $+\!\!+$ in Section 2.2 is to define them in terms of eliminators. Roughly, the eliminator for an F-algebra (A, c) is a term

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

with computation rule $\mathrm{elim}_F(P, \mathrm{step}_c, c(x)) = \mathrm{step}_c(x, \mathrm{dmap}_F(P, \mathrm{elim}(P, \mathrm{step}_c), x))$. Here, $\Box_F(P) : F(A) \to \mathrm{Set}$ is the type of inductive hypothesis for P; it consists of proofs that P holds at all F-substructures of x, and $\mathrm{dmap}_F(P) : (x : F(A) \to P(x)) \to (x : F(A)) \to \Box_F(P, x)$ takes care of recursive calls.

Example 3.1. Let $F(X) = \mathbf{1} + X$, i.e. F is the functor whose initial algebra is $(\mathbb{N}, [0, \mathrm{suc}])$. We then have

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

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

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

For polynomial functors F, $\Box_F$ can be defined inductively over the structure of F as is given in e.g. Dybjer and Setzer [8]. However, $\Box_F$ and dmap_F can be defined for any functor $F : \mathrm{Set} \to \mathrm{Set}$ by defining

$$\begin{aligned} \Box_F(P, x) &:= \{y : F(\Sigma\, z : A.\, P(z)) | F(\pi_0)(y) = x\} \\ \mathrm{dmap}_F(P, \mathrm{step}_c, x) &:= F(\lambda y.\langle y, \mathrm{step}_c(y)\rangle)(x) \enspace . \end{aligned}$$

We see that indeed $\Box_{\mathbf{1}+X}(P, \mathrm{inl}(\star)) \cong \mathbf{1}$ and $\Box_{\mathbf{1}+X}(P, \mathrm{inr}(n)) \cong P(n)$ as in Example 3.1.

3.2 The Generic Eliminator for an Inductive-Inductive Definition

Let us now generalise the preceding discussion from inductive definitions (i.e. endofunctors on Set) to inductive-inductive definitions (i.e. functors $\mathrm{Arg} = (\mathrm{Arg_A}, \mathrm{Arg_B})$ as in Definition 2.3). Since we replace the carrier set A with a carrier family (A, B), we should also replace the predicate $P : A \to \mathrm{Set}$ with a "predicate family" (P, Q) where $P : A \to \mathrm{Set}$ and $Q : (x : A) \to B(x) \to P(x) \to \mathrm{Set}$. This forces us to refine the step function $\mathrm{step}_c : (x : F(A)) \to \Box_F(P, x) \to P(c(x))$ into two functions

$$\begin{aligned} &\mathrm{step}_c : (x : \mathrm{Arg_A}(A, B)) \to \Box_{\mathrm{Arg_A}}(P, Q, x) \to P(c(x)) \enspace , \\ &\mathrm{step}_d : (x : \mathrm{Arg_A}(A, B)) \to (y : \mathrm{Arg_B}(A, B, c, x)) \to (\widetilde{x} : \Box_{\mathrm{Arg_A}}(P, Q, x)) \\ &\qquad\qquad \to \Box_{\mathrm{Arg_B}}(P, Q, c, \mathrm{step}_c, x, y, \widetilde{x}) \to Q(c(x), d(x, y), \mathrm{step}_c(x, \widetilde{x})) \enspace . \end{aligned}$$

As can already be seen in the types of step_c and step_d above, we replace $\Box_F$ with $\Box_{\mathrm{Arg_A}}$ and $\Box_{\mathrm{Arg_B}}$ of type

$$\begin{aligned} \Box_{\mathrm{Arg_A}}(P, Q) &: \mathrm{Arg_A}(A, B) \to \mathrm{Set} \enspace , \\ \Box_{\mathrm{Arg_B}}(P, Q) &: \big(\mathrm{step}_c : (x : \mathrm{Arg_A}(A, B)) \to \Box_{\mathrm{Arg_A}}(P, Q, x) \to P(c(x))\big) \to \\ &\quad (x : \mathrm{Arg_A}(A, B)) \to (y : \mathrm{Arg_B}(A, B, c, x)) \to \\ &\quad (\widetilde{x} : \Box_{\mathrm{Arg_A}}(P, Q, x)) \to \mathrm{Set} \end{aligned}$$

and we replace dmap_F with $\mathrm{dmap}_{\mathrm{Arg_A}}$, $\mathrm{dmap}_{\mathrm{Arg_B}}$ of type

$$\begin{aligned} \mathrm{dmap}_{\mathrm{Arg_A}}(P, Q) &: \big(f : (x : A) \to P(x)\big) \to \\ &\quad \big(g : (x : A) \to (y : B(x)) \to Q(x, y, f(x))\big) \to \\ &\quad (x : \mathrm{Arg_A}(A, B)) \to \Box_{\mathrm{Arg_A}}(P, Q, x) \\ \mathrm{dmap}_{\mathrm{Arg_B}}(P, Q) &: \big(\mathrm{step}_c : (x : \mathrm{Arg_A}(A, B)) \to \Box_{\mathrm{Arg_A}}(P, Q, x) \to P(c(x))\big) \to \\ &\quad \big(f : (x : A) \to P(x)\big) \to \\ &\quad \big(g : (x : A) \to (y : B(x)) \to Q(x, y, f(x))\big) \to \\ &\quad (x : \mathrm{Arg_A}(A, B)) \to (y : \mathrm{Arg_B}(A, B, c, x)) \\ &\qquad \to \Box_{\mathrm{Arg_B}}(P, Q, \mathrm{step}_c, x, y, \mathrm{dmap}_{\mathrm{Arg_A}}(P, Q, f, g, x)) \enspace . \end{aligned}$$

We can define $\Box_{\mathrm{Arg_A}}$, $\Box_{\mathrm{Arg_B}}$, $\mathrm{dmap}_{\mathrm{Arg_A}}$ and $\mathrm{dmap}_{\mathrm{Arg_B}}$ for arbitrary functors representing inductive-inductive definitions. First, define:

Definition 3.2. *Let $(A, B) \in Fam(Set)$, $P : A \to Set$, $Q : (x : A) \to B(x) \to P(x) \to Set$.*

(i) Define $\Sigma_{Fam(Set)}(A, B)\ (P, Q) \in Fam(Set)$ by

$$\Sigma_{Fam(Set)}(A, B)\ (P, Q) := (\Sigma\ A\ P, \lambda\langle a, p\rangle.\Sigma b{:}B(a).\ Q(a, b, p))$$

(ii) In addition, for $(f, g) : (A, B) \to (A', B')$ and

$$h : (x : A) \to P(f(x)) \qquad k : (x : A) \to (y : B(x)) \to Q(f(x), g(x, y), h(x)) \ ,$$

define $\langle(f, g),(h, k)\rangle : (A, B) \to \Sigma_{Fam(Set)}(A', B')\ (P, Q)$ by

$$\langle(f, g),(h, k)\rangle := (\lambda x.\langle f(x), h(x)\rangle, \lambda x\, y.\langle g(x, y), k(x, y)\rangle) \ .$$

(iii) For $h : (x : A) \to P(x)$ and $k : (x : A) \to (y : B(x)) \to Q(x, y, h(x))$, define $\overline{(h, k)} : (A, B) \to \Sigma_{Fam(Set)}(A, B)\ (P, Q)$ by $\overline{(h, k)} := \langle id, (h, k)\rangle$.

We have $(\pi_0, \pi_0') := (\pi_0, \lambda x.\pi_0) : \Sigma_{\mathrm{Fam(Set)}}(A, B)\ (P, Q) \to (A, B)$ with $(\pi_0, \pi_0')\circ \overline{(h, k)} = \mathrm{id}$. Note also that we can extend $\Sigma_{\mathrm{Fam(Set)}}$ to morphisms by defining $[(f, g),(h, k)] : \Sigma_{\mathrm{Fam(Set)}}(A, B)\ (P, Q) \to \Sigma_{\mathrm{Fam(Set)}}(A', B')\ (P', Q')$ for appropriate f, g, h, k by $[(f, g),(h, k)] = \langle(f, g) \circ (\pi_0, \pi_0'),(h, k)\rangle$. We can now define $\Box_{\mathrm{Arg_A}}$ and $\mathrm{dmap}_{\mathrm{Arg_A}}$:

Definition 3.3. *Define $\Box_{Arg_A}$ and $dmap_{Arg_A}$ with types as above by*

$$\Box_{Arg_A}(P, Q, x) := \{y : Arg_A(\Sigma_{Fam(Set)}(A, B)\ (P, Q))\ | Arg_A(\pi_0, \pi_0')(y) = x\} \ ,$$

$$dmap_{Arg_A}(P, Q, f, g) := Arg_A(\overline{(f, g)}) \ .$$

Note that we have an isomorphism

$$\varphi_{\mathrm{Arg_A}} : \mathrm{Arg_A}(\Sigma_{\mathrm{Fam(Set)}}(A, B)\ (P, Q)) \to \Sigma\, x{:}\mathrm{Arg_A}(A, B).\Box_{\mathrm{Arg_A}}(P, Q, x)$$

defined by $\varphi_{\mathrm{Arg_A}}(x) = \langle \mathrm{Arg_A}(\pi_0, \pi_0')(x), x\rangle$.

Definition 3.4. *Given P, Q, $step_c$, x, y, $\widetilde{x}$ as above, define*

(i) $\Sigma_{Dialg}(A, B, c)\ (P, Q, step_c) := (\Sigma_{Fam(Set)}\ (A, B)\ (P, Q), [c, step_c] \circ \varphi_{Arg_A})$,
(ii) $\Box_{Arg_B}(P, Q, step_c, x, y, \widetilde{x}) :=$
$\{z : Arg_B((\Sigma_{Dialg}(A, B, c)\ (P, Q, step_c)), \widetilde{x})\ |\ Arg_B(\pi_0, \pi_0', \widetilde{x}, z) = y\}$,
(iii) $dmap_{Arg_B}(P, Q, step_c, f, g) := Arg_B(\overline{(f, g)})$.

We can now define what the eliminators for inductive-inductive definitions are:

Definition 3.5. *We say that (A, B, c, d) in $\mathbb{E}_{Arg}$ has an eliminator, if there exist two terms*

$$\frac{\begin{array}{c} P : A \to Set \\ Q : (x : A) \to B(x) \to P(x) \to Set \\ step_c : (x : Arg_A(A, B)) \to \Box_{Arg_A}(P, Q, x) \to P(c(x)) \\ step_d : (x : Arg_A(A, B)) \to (y : Arg_B(A, B, c, x)) \to (\widetilde{x} : \Box_{Arg_A}(P, Q, x)) \\ \quad \to \Box_{Arg_B}(P, Q, c, step_c, x, y, \widetilde{x}) \to Q(c(x), d(x, y), step_c(x, \widetilde{x})) \end{array}}{\begin{array}{c} elim_{Arg_A}(P, Q, step_c, step_d) : (x : A) \to P(x) \\ elim_{Arg_B}(P, Q, step_c, step_d) : (x : A) \to (y : B(x)) \to Q(x, y, elim_{Arg_A}(P, Q, step_c, step_d, x)) \end{array}}$$

with

$$elim_{Arg_A}(P, Q, step_c, step_d, c(x)) = step_c(x, dmap'_{Arg_A})$$
$$elim_{Arg_B}(P, Q, step_c, step_d, c(x), d(x, y)) = step_d(x, y, dmap'_{Arg_A}, dmap'_{Arg_B})$$

where

$$dmap'_{Arg_A} = dmap_{Arg_A}(elim_{Arg_A}(P, Q, step_c, step_d), elim_{Arg_B}(P, Q, step_c, step_d), x)$$
$$dmap'_{Arg_B} = dmap_{Arg_B}(step_c, elim_{Arg_A}(P, Q, step_c, step_d), elim_{Arg_B}(P, Q, step_c, step_d), x, y) \ .$$

Example 3.6 (The eliminator for sorted lists). Recall from Example 2.5 that sorted lists were given by the functors $\mathrm{Arg}_{\mathrm{SList}}$, $\mathrm{Arg}_{\leqslant_{\mathrm{L}}}$, where

$$\mathrm{Arg}_{\mathrm{SList}}(A, B) = \mathbf{1} + (\Sigma\, n{:}\mathbb{N}.\, \Sigma\, \ell{:}A.\, B(n, \ell))$$

Thus, we see that e.g.

$$\Box_{\mathrm{Arg}_{\mathrm{SList}}}(P, Q, \mathrm{inl}(\star)) = \{y : \mathbf{1} + \ldots \mid (\mathrm{id} + \ldots)(y) = \mathrm{inl}(\star)\} \cong \mathbf{1}$$
$$\Box_{\mathrm{Arg}_{\mathrm{SList}}}(P, Q, \mathrm{inr}(\langle n, \ell, p\rangle)) \cong$$
$$\{y : \Sigma\, n'{:}\mathbb{N}.\, \Sigma\, \langle \ell', \widetilde{\ell}\rangle{:}(\Sigma A P).\, \Sigma p'{:}B(n, \ell).\, Q(n', \ell', p', \widetilde{\ell}) \mid \Sigma(\mathrm{id}, \Sigma(\pi_0, \pi'_0))(y) = \langle n, \ell, p\rangle\}$$
$$\cong \Sigma\widetilde{\ell}{:}P(\ell).\, Q(n, \ell, p, \widetilde{\ell})$$

and similarly for $\Box_{\mathrm{Arg}_{\leqslant_{\mathrm{L}}}}$, so that the eliminators are equivalent to

$$\begin{aligned}
\mathrm{elim}_{\mathrm{SortedList}} : {} & (P : \mathrm{SortedList} \to \mathrm{Set}) \to \\
& (Q : (n : \mathbb{N}) \to (\ell : \mathrm{SortedList}) \to n \leqslant_{\mathrm{L}} \ell \to P(\ell) \to \mathrm{Set}) \to \\
& (\mathrm{step}_{\mathrm{nil}} : P(\mathrm{nil})) \to \\
& \big(\mathrm{step}_{\mathrm{cons}} : (n : \mathbb{N}) \to (\ell : \mathrm{SortedList}) \to (p : n \leqslant_{\mathrm{L}} \ell) \to (\widetilde{\ell} : P(\ell)) \\
& \quad \to Q(n, \ell, p, \widetilde{\ell}) \to P(\mathrm{cons}(n, \ell, p))\big) \to \\
& \big(\mathrm{step}_{\mathrm{triv}} : (n : \mathbb{N}) \to Q(n, \mathrm{nil}, \mathrm{triv}_n, \mathrm{step}_{\mathrm{nil}})\big) \to \\
& \big(\mathrm{step}_{\ll\cdot\gg} : (m : \mathbb{N}) \to (n : \mathbb{N}) \to (\ell : \mathrm{SortedList}) \to (p : n \leqslant_{\mathrm{L}} \ell) \\
& \quad \to (q : m \leqslant n) \to (p' : m \leqslant_{\mathrm{L}} \ell) \to (\widetilde{\ell} : P(\ell)) \\
& \quad \to (\widetilde{p} : Q(n, \ell, p, \widetilde{\ell})) \to (\widetilde{p'} : Q(m, \ell, p', \widetilde{\ell})) \\
& \quad \to Q(m, \mathrm{cons}(n, \ell, p), \ll q, p' \gg_{p,m,n,\ell}, \mathrm{step}_{\mathrm{cons}}(n, \ell, p, \widetilde{\ell}, \widetilde{p}))\big) \to \\
& (\ell : \mathrm{SortedList}) \to P(\ell) \ , \\
\mathrm{elim}_{\leqslant_{\mathrm{L}}} : {} & \ldots \to \\
& (n : \mathbb{N}) \to (\ell : \mathrm{SortedList}) \to (p : n \leqslant_{\mathrm{L}} \ell) \\
& \quad \to Q(n, \ell, p, \mathrm{elim}_{\mathrm{SortedList}}(P, Q, \mathrm{step}_{\mathrm{nil}}, \mathrm{step}_{\mathrm{cons}}, \mathrm{step}_{\mathrm{triv}}, \mathrm{step}_{\ll\cdot\gg}, \ell)) \ .
\end{aligned}$$

3.3 The Equivalence Between Having an Eliminator and Being Initial

We now prove the promised equivalence. In what follows, let $\mathrm{Arg} = (\mathrm{Arg}_{\mathrm{A}}, \mathrm{Arg}_{\mathrm{B}})$ be functors for an inductive-inductive definition.

Lemma 3.7. *There is an isomorphism*

$$\begin{aligned}\varphi_{Arg} = (\varphi_{Arg_A}, \varphi_{Arg_B}) : Arg(\Sigma_{Dialg}(A, B, c)\ (P, Q, step_c)) \\ \to \Sigma_{Fam(Set)} Arg(A, B, c)\ (\Box_{Arg}(P, Q, step_c))\end{aligned}$$

such that $(\pi_0, \pi_0') \circ \varphi_{Arg} = Arg(\pi_0, \pi_0')$ *and*

$$\varphi_{Arg} \circ Arg(\overline{(f, g)}) = \overline{(dmap_{ArgA}(P, Q, f, g), dmap_{Arg_B}(P, Q, step_c, f, g))}\ . \qquad \Box$$

Proposition 3.8. *Every initial object* (A, B, c, d) *in* $\mathbb{E}_{Arg}$ *has an eliminator.*

Proof. Let P, Q, step_c, step_d as in the type signature for $\mathrm{elim}_{\mathrm{Arg_A}}$ and $\mathrm{elim}_{\mathrm{Arg}}$ be given. Define $\mathrm{in}_\Sigma : \mathrm{Arg}(\Sigma_{\mathrm{Dialg}}(A, B, c)\ (P, Q, \mathrm{step}_c)) \to V(\Sigma_{\mathrm{Dialg}}(A, B, c)\ (P, Q, \mathrm{step}_c))$ by $\mathrm{in}_\Sigma = [(c, d), (\mathrm{step}_c, \mathrm{step}_d)] \circ \varphi_{\mathrm{Arg}}$. This makes $\Sigma_{\mathrm{Dialg}}(A, B, c)\ (P, Q, \mathrm{step}_c)$ an object of $\mathbb{E}_{\mathrm{Arg}}$.

Since (A, B, c, d) is initial in $\mathbb{E}_{\mathrm{Arg}}$, we get a morphism $(h, h') : (A, B) \to \Sigma_{\mathrm{Fam(Set)}}(A, B)\ (P, Q)$ which makes the top part of the following diagram commute:

$$\begin{array}{ccccc}
\mathrm{Arg}(A,B,c) & & \xrightarrow{(c,d)} & & (A,B) \\
\downarrow{\scriptstyle \mathrm{Arg}(h,h')} & & & & \downarrow{\scriptstyle (h,h')} \\
\mathrm{Arg}(\Sigma(A,B,c)\ (P,Q,\mathrm{step}_c)) & \xrightarrow{\varphi_{\mathrm{Arg}}} & \Sigma\mathrm{Arg}(A,B,c)\ (\Box(P,Q,\mathrm{step}_c)) & \xrightarrow{[(c,d),(\mathrm{step}_c,\mathrm{step}_d)]} & \Sigma(A,B)\ (P,Q) \\
\downarrow{\scriptstyle \mathrm{Arg}(\pi_0,\pi_0')} & \swarrow{\scriptstyle (\pi_0,\pi_0')} & & & \downarrow{\scriptstyle (\pi_0,\pi_0')} \\
\mathrm{Arg}(A,B,c) & & \xrightarrow{(c,d)} & & (A,B)
\end{array}$$

The bottom part commutes by Lemma 3.7 and calculation. Hence $(\pi_0, \pi_0') \circ (h, h')$ is a morphism in $\mathbb{E}_{\mathrm{Arg}}$ and we must have $(\pi_0, \pi_0') \circ (h, h') = \mathrm{id}$ by initiality. Thus $\pi_1 \circ h : (x : A) \to P(x)$ and $\pi_1(h'(x, y)) : Q(x, y, \pi_1(h(x)))$ for $x : A$, $y : B(x)$, so we can define $\mathrm{elim}_{\mathrm{Arg_A}}(P, Q, \mathrm{step}_c, \mathrm{step}_d) = \pi_1 \circ h$ and $\mathrm{elim}_{\mathrm{Arg_B}}(P, Q, \mathrm{step}_c, \mathrm{step}_d, x, y) = \pi_1(h'(x, y))$.

To verify the computation rules, note that since $(\pi_0, \pi_0') \circ (h, h') = \mathrm{id}$, we have $(h, h') = \overline{(\pi_1, \pi_1') \circ (h, h')}$. We only show the calculation for $\mathrm{Arg_A}$:

$$\begin{aligned}
\mathrm{elim}_{\mathrm{Arg_A}}(P, Q, \mathrm{step}_c, \mathrm{step}_d, c(x))) &= \pi_1(h(c(x))) \\
&= \mathrm{step}_c(\varphi_{\mathrm{Arg_A}}(\mathrm{Arg_A}(h, h')(x))) \\
&= \mathrm{step}_c(\varphi_{\mathrm{Arg_A}}(\mathrm{Arg_A}(\overline{(\pi_1, \pi_1') \circ (h, h')})(x))) \\
&= \mathrm{step}_c(x, \mathrm{dmap}_{\mathrm{Arg_A}}((\pi_1, \pi_1') \circ (h, h'))(x)) \\
&= \mathrm{step}_c(x, \mathrm{dmap}'_{\mathrm{Arg_A}})
\end{aligned} \qquad \Box$$

Proposition 3.9. *Every* (A, B, c, d) *with an eliminator is weakly initial in* $\mathbb{E}_{Arg}$.

Proof. Let (A', B', c', d') be another object in $\mathbb{E}_{\mathrm{Arg}}$. Notice that for $P(x) = A'$, $Q(x, y, \tilde{x}) = B'(\tilde{x})$, the usually dependent second projections π_1, π_1' become non-dependent and make up a morphism $(\pi_1, \pi_1') : \Sigma_{\mathrm{Fam(Set)}}(A, B)\ (P, Q) \to (A', B')$. Since

$$\pi_1 \circ [c, c' \circ \mathrm{Arg_A}(\pi_1, \pi_1') \circ \varphi^{-1}_{\mathrm{Arg_A}}] \circ \varphi_{\mathrm{Arg_A}} = c' \circ \mathrm{Arg_A}(\pi_1, \pi_1')\ ,$$

this lifts to $(\pi_1, \pi_1') : \Sigma_{\mathrm{Dialg}}(A, B, c)\ (P, Q, c' \circ \mathrm{Arg}_{\mathrm{A}}(\pi_1, \pi_1') \circ \varphi^{-1}_{\mathrm{Arg}_{\mathrm{A}}}) \to (A', B', c')$. By currying $(f, g) \coloneqq (c', d') \circ \mathrm{Arg}(\pi_1, \pi_1') \circ \varphi^{-1}_{\mathrm{Arg}}$, we get

$$\begin{aligned}
&\widehat{f} : (x : \mathrm{Arg}_{\mathrm{A}}(A, B)) \to \Box_{\mathrm{Arg}_{\mathrm{A}}}(P, Q, x) \to A' \\
&\widehat{g} : (x : \mathrm{Arg}_{\mathrm{A}}(A, B)) \to (y : \mathrm{Arg}_{\mathrm{B}}(A, B, c, x)) \to (\widetilde{x} : \Box_{\mathrm{Arg}_{\mathrm{A}}}(P, Q, x)) \\
&\qquad\qquad \to \Box_{\mathrm{Arg}_{\mathrm{B}}}(P, Q, c, \widehat{f}, x, y, \widetilde{x}) \to B'(\widehat{f}(x, \widetilde{x}))
\end{aligned}$$

so that $(h, h') \coloneqq (\mathrm{elim}_{\mathrm{Arg}_{\mathrm{A}}}(P, Q, \widehat{f}, \widehat{g}), \mathrm{elim}_{\mathrm{Arg}_{\mathrm{B}}}(P, Q, \widehat{f}, \widehat{g})) : (A, B) \to (A', B')$.

We have to check that $(h, h') \circ (c, d) = (c', d') \circ \mathrm{Arg}(h, h')$.

$$\begin{aligned}
(h, h') \circ (c, d) &= (\mathrm{elim}_{\mathrm{Arg}_{\mathrm{A}}}(P, Q, \widehat{f}, \widehat{g}), \mathrm{elim}_{\mathrm{Arg}_{\mathrm{B}}}(P, Q, \widehat{f}, \widehat{g})) \circ (c, d) \\
&= (\widehat{f}, \widehat{g}) \circ \overline{(\mathrm{dmap}_{\mathrm{Arg}_{\mathrm{A}}}(h, h'), \mathrm{dmap}_{\mathrm{Arg}_{\mathrm{B}}}(h, h'))} \\
&= (\widehat{f}, \widehat{g}) \circ \varphi_{\mathrm{Arg}} \circ \mathrm{Arg}(\overline{h, h'}) \\
&= (c', d') \circ \mathrm{Arg}(\pi_1, \pi_1') \circ \mathrm{Arg}(\overline{h, h'}) \\
&= (c', d') \circ \mathrm{Arg}(h, h') \qquad \Box
\end{aligned}$$

For strictly positive functors, we can say more, since we can argue by induction over their construction:

Theorem 3.10. *The functors $Arg_\gamma = (Arg_{\gamma_A}, Arg_{\gamma_B})$ from the original axiomatisation as described in Section 2.3 have eliminators if and only if $\mathbb{E}_{Arg_\gamma}$ has an initial object.*

Proof. Putting Proposition 3.8 and Proposition 3.9 together, all that is left to prove is that given an eliminator, the arrow (h, h') we construct is actually unique. Assume that (k, k') is another arrow with $(k, k') \circ (c, d) = (c', d') \circ \mathrm{Arg}_\gamma(k, k')$.

We use the eliminator (and extensional equality) to prove that $(h, h') = (k, k')$; let $P(x) = (h(x) = k(x))$ and $Q(x, y, \widetilde{x}) = (h'(x, y) = k'(x, y))$. It is enough to prove $P(c(x))$ and $Q(c(x), d(x, y), _)$ for arbitrary $x : \mathrm{Arg}_{\gamma_{\mathrm{A}}}(A, B)$, $y : \mathrm{Arg}_{\gamma_{\mathrm{B}}}(A, B, c, x)$, given the induction hypothesis $\Box_{\mathrm{Arg}_{\mathrm{A}}}(P, Q)$ and $\Box_{\mathrm{Arg}_{\mathrm{B}}}(P, Q)$. By induction on the buildup of $\mathrm{Arg}_{\gamma_{\mathrm{A}}}$ and $\mathrm{Arg}_{\gamma_{\mathrm{B}}}$, we can prove that $\Box_{\mathrm{Arg}_{\mathrm{A}}}(P, Q)$ and $\Box_{\mathrm{Arg}_{\mathrm{B}}}(P, Q)$ give that $\mathrm{Arg}(h, h') = \mathrm{Arg}(k, k')$, and hence

$$(h, h') \circ (c, d) = (c', d') \circ \mathrm{Arg}(h, h') = (c', d') \circ \mathrm{Arg}(k, k') = (k, k') \circ (c, d) \ .$$

Using the elimination principle, we conclude that $(h, h') = (k, k')$. □

4 Conclusions and Future Work

We have shown how to give a categorical semantics for inductive-inductive definitions, a principle for defining data types in Martin-Löf Type Theory. In order to do this, we generalised the usual initial algebra semantics to a dialgebra setting and showed that there is still an equivalence between this semantics and the more traditional formulation in terms of elimination and computation rules.

Future work includes extending the notion of containers [1] to inductive-inductive definitions. We also conjecture that W-types are enough to ensure the existence of inductive-inductive definitions in an extensional theory. More precisely, it should be possible to interpret inductive-inductive definitions as indexed inductive definitions, for which W-types are enough [2].

It could also be worthwhile to generalise this work to a unified setting including other forms of inductive definitions: let $F, G : \mathbb{C} \to \mathbb{D}$ be functors between categories having all finite limits. One can then extend $\mathbb{C}$ and $\mathbb{D}$ to Categories with Families [6, 13] and use that structure to define the concept of an eliminator for F and G. If G is left exact, one can show that having an eliminator and being initial in (a subcategory of) $\mathrm{Dialg}(F, G)$ is equivalent.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theoretical Computer Science 342(1), 3 – 27 (2005)
2. Altenkirch, T., Morris, P.: Indexed containers. In: Logic In Computer Science, 2009. LICS '09. 24th Annual IEEE Symposium on. pp. 277 –285 (2009)
3. Chapman, J.: Type theory should eat itself. Electronic Notes in Theoretical Computer Science 228, 21–36 (2009)
4. Danielsson, N.A.: A formalisation of a dependently typed language as an inductive-recursive family. Lecture Notes in Computer Science 4502, 93–109 (2007)
5. Dybjer, P.: Inductive families. Formal aspects of computing 6(4), 440–465 (1994)
6. Dybjer, P.: Internal type theory. LNCS 1158, 120–134 (1996)
7. Dybjer, P., Setzer, A.: A finite axiomatization of inductive-recursive definitions. In: TLCA'99. pp. 129–146 (1999)
8. Dybjer, P., Setzer, A.: Induction–recursion and initial algebras. Annals of Pure and Applied Logic 124(1-3), 1–47 (2003)
9. Ghani, N., Johann, P., Fumex, C.: Fibrational induction rules for initial algebras. In: Computer Science Logic. LNCS, vol. 6247, pp. 336–350. Springer (2010)
10. Goguen, J., Thatcher, J., Wagner, E., Wright, J.: Initial algebra semantics and continuous algebras. Journal of the ACM 24(1), 68–95 (1977)
11. Hagino, T.: A Categorical Programming Language. Ph.D. thesis, University of Edinburgh (1987)
12. Hermida, C., Jacobs, B.: Structural induction and coinduction in a fibrational setting. Information and Computation 145(2), 107 – 152 (1998)
13. Hofmann, M.: Syntax and semantics of dependent types. In: Semantics and Logics of Computation, pp. 79 – 130. Cambridge University Press (1997)
14. Martin-Löf, P.: Intuitionistic type theory. Bibliopolis Naples (1984)
15. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's type theory: an introduction. Oxford University Press (1990)
16. Nordvall Forsberg, F., Setzer, A.: Inductive-inductive definitions. In: Computer Science Logic. LNCS, vol. 6247, pp. 454–468. Springer (2010)
17. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
18. Poll, E., Zwanenburg, J.: From algebras and coalgebras to dialgebras. Electronic Notes in Theoretical Computer Science 44(1), 289–307 (2001)