# Inductive-Inductive Definitions

Fredrik Nordvall Forsberg[⋆] and Anton Setzer[⋆]

Swansea University
{csfnf,a.g.setzer}@swansea.ac.uk

**Abstract.** We present a principle for introducing new types in type theory which generalises strictly positive indexed inductive data types. In this new principle a set $A$ is defined inductively simultaneously with an $A$-indexed set $B$, which is also defined inductively. Compared to indexed inductive definitions, the novelty is that the index set $A$ is generated inductively simultaneously with $B$. In other words, we mutually define two inductive sets, of which one depends on the other.

Instances of this principle have previously been used in order to formalise type theory inside type theory. However the consistency of the framework used (the theorem prover Agda) is not so clear, as it allows the definition of a universe containing a code for itself. We give an axiomatisation of the new principle in such a way that the resulting type theory is consistent, which we prove by constructing a set-theoretic model.

## 1 Introduction

Martin-Löf Type Theory [12] is a foundational framework for constructive mathematics, where induction plays a major part in the construction of sets. Martin-Löf's formulation [12] includes inductive definitions of for example Cartesian products, disjoint unions, the identity set, finite sets, the natural numbers, well-orderings and lists. External schemas for general inductive sets and inductive families have been given by Backhouse et. al. [2] and Dybjer [6] respectively. Indexed inductive definitions have also been used for generic programming in dependent type theory [3,13].

Another induction principle is *induction-recursion*, where a set $U$ is constructed inductively simultaneously with a recursively defined function $T : U \to D$ for some possibly large type $D$. The constructor for $U$ may depend negatively on $T$ applied to elements of $U$. The main example is Martin-Löf's universe à la Tarski [15]. Dybjer [8] gave a schema for such inductive-recursive definitions, and this has been internalised by Dybjer and Setzer [9,10,11].

In this article, we present another induction principle, which we, in reference to induction-recursion, call *induction-induction*. A set $A$ is inductively defined simultaneously with an $A$-indexed set $B$, which is also inductively defined, and

⋆ Supported by EPSRC grant EP/G033374/1, Theory and applications of induction-recursion.

the introduction rules for $A$ may also refer to $B$. So we have formation rules $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$ and typical introduction rules might take the form

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

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

Let us consider a first example, which will serve as a running example to illustrate the formal rules. We simultaneously define a set of platforms together with buildings constructed on these platforms. The ground is a platform, and if we have a building, we can always construct a new platform from it by building an extension. We can always build a building on top of any platform, and if we have an extension, we can also construct a building hanging from it. See the extended version [14] of this article for an illustration.) This gives rise to the following inductive-inductive definition of Platform : Set, Building : Platform → Set (where $p$ : Platform means that $p$ is a platform and $b$ : Building($p$) means that $b$ is a building constructed on the platform $p$)[1] with constructors

$$\begin{aligned}
\mathrm{ground} &: \mathrm{Platform} \ , \\
\mathrm{extension} &: ((p : \mathrm{Platform}) \times \mathrm{Building}(p)) \to \mathrm{Platform} \ , \\
\mathrm{onTop} &: (p : \mathrm{Platform}) \to \mathrm{Building}(p) \ , \\
\mathrm{hangingUnder} &: ((p : \mathrm{Platform}) \times (b : \mathrm{Building}(p))) \to \mathrm{Building}(\mathrm{extension}(\langle p, b \rangle)) .
\end{aligned}$$

Note that the index of the codomain of hangingUnder is $\mathrm{extension}(\langle p, b \rangle)$, i.e. $\mathrm{hangingUnder}(\langle p, b \rangle) : \mathrm{Building}(\mathrm{extension}(\langle p, b \rangle))$. In other words, it is not possible to have a building hanging under the ground.

Inductive-inductive definitions have been used by Dybjer [7], Danielsson [5] and Chapman [4] to internalise the syntax and semantics of type theory. Slightly simplified, they define a set Ctxt of contexts, a family Ty : Ctxt → Set of types in a given context, and a family Term : ($\Gamma$ : Ctxt) → Ty($\Gamma$) → Set of terms of a given type. Let us for simplicity only consider contexts and types. The set Ctxt of contexts has two constructors

$$\begin{aligned}
\varepsilon &: \mathrm{Ctxt} \ , \\
\mathrm{cons} &: ((\Gamma : \mathrm{Ctxt}) \times \mathrm{Ty}(\Gamma)) \to \mathrm{Ctxt} \ ,
\end{aligned}$$

corresponding to the empty context and extending a context $\Gamma$ with a new type. In our simplified setting, Ty : Ctxt → Set has the following constructors

$$\begin{aligned}
\text{'set'} &: (\Gamma : \mathrm{Ctxt}) \to \mathrm{Ty}(\Gamma) \ , \\
\Pi &: ((\Gamma : \mathrm{Ctxt}) \times (A : \mathrm{Ty}(\Gamma)) \times \mathrm{Ty}(\mathrm{cons}(\langle \Gamma, A \rangle))) \to \mathrm{Ty}(\Gamma) \ .
\end{aligned}$$

[1] The collection of small types in Martin-Löf type theory is called Set for historic reasons, whereas Type is reserved for the collection of large types. The judgement Building : Platform → Set means that for every $p$ : Platform, Building($p$) : Set. The type theoretic notation will be further explained in Section 2.

The first constructor states that 'set' is a type in any context. The second constructor $\Pi$ is the constructor for the $\Pi$-type: If we have a type $A$ in a context $\Gamma$, and another type $B$ in $\Gamma$ extended by $A$ (corresponding to abstracting a variable of type $A$), then $\Pi(A, B)$ is also a type in $\Gamma$.

Note how the constructor cons for Ctxt has an argument of type $\mathrm{Ty}(\Gamma)$, even though Ty is indexed by Ctxt. It is also worth noting that $\Pi$ has an argument of type $\mathrm{Ty}(\mathrm{cons}(\langle \Gamma, A\rangle))$, i.e. we are using the constructor for Ctxt in the index of Ty. In general, we could of course imagine an argument of type $\mathrm{Ty}(\mathrm{cons}(\langle \mathrm{cons}(\langle \Gamma, A\rangle), A'\rangle))$ etc.

Both Danielsson [5] and Chapman [4] have used the proof assistant Agda [17] as a framework for their formalisation. Agda supports inductive-inductive (and inductive-recursive) definitions via the `mutual` keyword. However, the theory behind Agda is unclear, especially in relation to mutual definitions. For example, Agda allows the definition of a universe $U$ à la Tarski, with a code $u : U$ for itself, i.e. $T(u) = U$. This does not necessarily mean that Agda is inconsistent by Girard's paradox, as, if we also demand closure under $\Pi$ or $\Sigma$, the positivity checker rejects the code. However the consistency of Agda is by no means clear.

Nevertheless, Agda is an excellent tool for trying out ideas. We have formalised our theory in Agda, and this formalisation can be found on the authors' home pages, together with an extended version of this article [14], containing proofs and details that have been omitted due to space constraints.

## 2 Type Theoretic Preliminaries

We work in a type theory with at least two universes Set and Type, with Set : Type and if $A$ : Set then $A$ : Type. Both Set and Type are closed under dependent function types, written $(x : A) \to B$ (sometimes denoted $(\Pi x : A)B$), where $B$ is a set or type depending on $x : A$. Abstraction is written as $\lambda x : A.e$, where $e : B$ depending on $x : A$, and application as $f(x)$. Repeated abstraction and application are written as $\lambda x_1 : A_1 \ldots x_k : A_k.e$ and $f(x_1, \ldots, x_k)$. If the type of $x$ can be inferred, we simply write $\lambda x.e$ as an abbreviation. Furthermore, both Set and Type are closed under dependent sums, written $(x : A) \times B$ (sometimes denoted $(\Sigma x : A)B$), where $B$ is a set or type depending on $x : A$, with pairs $\langle a, b\rangle$, where $a : A$ and $b : B[x := a]$. We also have $\beta$- and $\eta$-rules for both dependent function types and sums.

We need an empty type $\mathbf{0}$ : Set, with elimination $!_A : (x : \mathbf{0}) \to A$ for every $A : \mathbf{0} \to \mathrm{Set}$. We need a unit type $\mathbf{1}$ : Set, with unique element $\star : \mathbf{1}$. We include an $\eta$-rule stating that if $x : \mathbf{1}$, then $x = \star : \mathbf{1}$. Moreover, we include a two element set $\mathbf{2}$ : Set, with elements $\mathrm{tt} : \mathbf{2}$, $\mathrm{ff} : \mathbf{2}$ and elimination constant $\mathbf{if} \cdot \mathbf{then} \cdot \mathbf{else} \cdot : (a : \mathbf{2}) \to A(\mathrm{tt}) \to A(\mathrm{ff}) \to A(a)$ where $A(i)$ : Type for $i : \mathbf{2}$.

With $\mathbf{if} \cdot \mathbf{then} \cdot \mathbf{else} \cdot$ and dependent products, we can now define the disjoint union of two sets $A + B := (x : \mathbf{2}) \times (\mathbf{if}\ x\ \mathbf{then}\ A\ \mathbf{else}\ B)$ with constructors $\mathrm{inl} = \lambda a : A.\langle \mathrm{tt}, a\rangle$ and $\mathrm{inr} = \lambda b : B.\langle \mathrm{ff}, b\rangle$, and prove the usual formation, introduction, elimination and equality rules. We write $A_0 + A_1 + \ldots + A_n$ for $A_0 + (A_1 + (\ldots + A_n)\cdots)$ and $\mathrm{in}_k(a)$ for the $k$th injection $\mathrm{inl}(\mathrm{inr}^k(a))$ (with special case $\mathrm{in}_n(a) = \mathrm{inr}^n(a)$).

Using (the derived) elimination rules for +, we can, for $A, B : \mathrm{Set}$, $C : A + B \to \mathrm{Type}$ and $f : (a : A) \to C(\mathrm{inl}(a))$, $g : (b : B) \to C(\mathrm{inr}(b))$, define the case distinction $f \sqcup g : (c : A + B) \to C(c)$ with equality rules

$$(f \sqcup g)(\mathrm{inl}(a)) = f(a) \ ,$$
$$(f \sqcup g)(\mathrm{inr}(b)) = g(b) \ .$$

We will use the same notation even when $C$ does not depend on $c : A+B$, and we will write $f \parallel g : A+B \to C+D$, where $f : A \to C$, $g : B \to D$, for $(\mathrm{inl} \circ f) \sqcup (\mathrm{inr} \circ g)$.

Intensional type theory in Martin-Löf's logical framework extended with dependent products and **0**, **1** and **2** has all the features we need. Thus, our development could, if one so wishes, be seen as an extension of the logical framework.

## 3 From Inductive to Inductive-Inductive Definitions

Let us first, before we move on to inductive-inductive definitions, informally consider how to formalise a simultaneous (generalised) inductive definition of two sets $A$ and $B$, given by constructors

$$\mathrm{intro_A} : \Phi_\mathrm{A}(A, B) \to A \quad \mathrm{intro_B} : \Phi_\mathrm{B}(A, B) \to B$$

where $\Phi_\mathrm{A}$ and $\Phi_\mathrm{B}$ are strictly positive in the following sense:

- The constant $\Phi(A, B) = \mathbf{1}$ is strictly positive. It corresponds to an introduction rule with no arguments (or more precisely, the trivial argument $x : \mathbf{1}$).
- If $K$ is a set and $\Psi_x$ is strictly positive, depending on $x : K$, then $\Phi(A, B) = (x : K) \times \Psi_x(A, B)$, corresponding to the addition of a non-inductive premise, is strictly positive. So $\mathrm{intro}_A$ has one non-inductive argument $x : K$, followed by the arguments given by $\Psi_x(A, B)$.
- If $K$ is a set and $\Psi$ is strictly positive, then $\Phi(A, B) = (K \to A) \times \Psi(A, B)$ is strictly positive. This corresponds to the addition of a premise inductive in $A$, where $K$ corresponds to the hypothesis of this premise in a generalised inductive definition. So $\mathrm{intro}_A$ has one inductive argument $f : K \to A$, followed by the arguments given by $\Psi(A, B)$.
- Likewise, if $K$ is a set and $\Psi$ is strictly positive, then $\Phi(A, B) = (K \to B) \times \Psi(A, B)$ is strictly positive. This is similar to the previous case.

In an inductive-inductive definition, $B$ is indexed by $A$, so the constructor for $B$ is replaced by

$$\mathrm{intro_B} : (a : \Phi_\mathrm{B}(A, B)) \to B(i_{A,B}(a))$$

for some index $i_{A,B}(a) : A$ which might depend on $a : \Phi_\mathrm{B}(A, B)$. Furthermore, we must modify the inductive case for $B$ to specify an index as well. This index can (and usually does) depend on earlier inductive arguments, so that the new inductive cases become

- If $K$ is a set, and $\Psi_f$ is strictly positive, depending[2] on $f : K \to A$ only in indices for $B$, then $\Phi(A, B) = (f : K \to A) \times \Psi_f(A, B)$ is strictly positive.
- If $K$ is a set, $i_{A,B} : K \to A$ is a function and $\Psi_f$ is strictly positive, depending on $f : (x : K) \to B(i_{A,B}(x))$ only in indices for $B$, then $\Phi(A, B) = (f : ((x : K) \to B(i_{A,B}(x)))) \times \Psi_f(A, B)$ is strictly positive.

In what way can the index depend on $f$? Before we know the constructor for $A$, we do not know any functions with codomain $A$, so the index can only depend directly on $f$ (e.g. $B(f(x))$). When we define the constructor for $B$, the situation is similar, but now we know one function into $A$, namely $\text{intro}_{\text{A}} : \Phi_{\text{A}}(A, B) \to A$, so that the index could be e.g. $\text{intro}_{\text{A}}(f(x), b)$. (Our approach could also be straightforwardly extended to allow several constructors for $A$, where later constructors make use of earlier ones.)

## 4 An Axiomatisation

We proceed as in Dybjer and Setzer [9] and introduce a datatype of codes for constructors. In other words, we define a type SP (for strictly positive) whose elements represent the inductively defined sets, together with a way to construct the real sets from the representing codes. However, as we have two sets $A$ and $B$ with different roles, we need two types $\text{SP}_{\text{A}}$ and $\text{SP}_{\text{B}}$ of codes.

What do we need to know in order to reconstruct the inductively defined sets? A moment's thought shows that all we need is the domain of the constructors, and in the case of $B$, we also need the index of the codomain of the constructor. From this, we can write down the introduction rules, and the elimination rules should be determined by these (see e.g. [6]). Thus, the codes in $\text{SP}_{\text{A}}$ and $\text{SP}_{\text{B}}$ will be codes for the domain of the constructors, and we will have functions $\text{Arg}_{\text{A}}$, $\text{Arg}_{\text{B}}$ that map the code to the domain it represents. For $B$, there will also be a function $\text{Index}_{\text{B}}$ that gives the index (in $A$) of the codomain of the constructor.

We will have special codes A-ind, B-ind for arguments that are inductive in $A$ and $B$ respectively. In the case of B-ind, we also need to specify an index, which might depend on earlier arguments. For instance, the index $p$ of the type of the second argument for the extension constructor

$$\text{extension} : ((p : \text{Platform}) \times \text{Building}(p)) \to \text{Platform}$$

depends on the first argument. How can we specify this index in the code? We cannot make use of the sets $A$ and $B$ themselves, since they are to be defined, but we can refer to their existence. We will introduce parameters $A_{\text{ref}}$, $B_{\text{ref}}$ that get updated during the construction of the code. $A_{\text{ref}}$ determines the elements of $A$ that we can refer to, and $B_{\text{ref}}$ determines the elements $b$ of $B$, together with the index $a$ such that $b : B(a)$. At the beginning, we cannot refer to any arguments, and so $A_{\text{ref}} = B_{\text{ref}} = \mathbf{0}$. For example, after having written down the first argument $p : \text{Platform}$, $A_{\text{ref}}$ would be extended to include also an element representing $p$, which we use when writing down the type of the second argument, $\text{Building}(p)$.

[2] The somewhat vague and informal phrase "depending on $f$ only in indices for $B$" will be given an exact meaning in the formalisation in the next section.

## 4.1 $\mathbf{SP_A}$ and $\mathbf{Arg_A}$

The above discussion leads us to the following formation rule for $\mathrm{SP_A}$:

$$\frac{A_{\mathrm{ref}} : \mathrm{Set} \quad B_{\mathrm{ref}} : \mathrm{Set}}{\mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}) : \mathrm{Type}}$$

$A_{\mathrm{ref}}$ and $B_{\mathrm{ref}}$ can be any sets. The codes for inductive-inductive definitions will however be elements from $\mathrm{SP}'_{\mathrm{A}} := \mathrm{SP_A}(\mathbf{0}, \mathbf{0})$, i.e. codes that do not refer to any elements to start with.

The introduction rules for $\mathrm{SP_A}$ reflect the rules for strict positivity in Section 3. The rules are as follows (we suppress the global premise $A_{\mathrm{ref}}, B_{\mathrm{ref}} : \mathrm{Set}$):

$$\frac{}{\mathrm{nil_A} : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}})} \qquad \frac{K : \mathrm{Set} \quad \gamma : K \to \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}})}{\mathrm{nonind}(K, \gamma) : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}})}$$

$$\frac{K : \mathrm{Set} \quad \gamma : \mathrm{SP_A}(A_{\mathrm{ref}} + K, B_{\mathrm{ref}})}{\text{A-ind}(K, \gamma) : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}})} \quad \frac{K : \mathrm{Set} \quad h_{\mathrm{index}} : K \to A_{\mathrm{ref}} \quad \gamma : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}} + K)}{\text{B-ind}(K, h_{\mathrm{index}}, \gamma) : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}})}$$

The code $\mathrm{nil_A}$ represents a trivial constructor (the base case). The code $\mathrm{nonind}(K, \gamma)$ is meant to represent a noninductive argument $x : K$, with the rest of the arguments given by $\gamma(x)$. The code $\text{A-ind}(K, \gamma)$ is meant to represent a (generalised) inductive argument of type $K \to A$, with the rest of the arguments given by $\gamma$. Finally, the code $\text{B-ind}(K, h_{\mathrm{index}}, \gamma)$ represents an inductive argument of type $(x : K) \to B(i(x))$, where the index $i(x)$ is determined by $h_{\mathrm{index}}$, and the rest of the arguments are given by $\gamma$. For instance, a constructor

$$c : ((x : \mathbf{2}) \times (\mathbb{N} \to A)) \to A$$

has the code $\gamma_c = \mathrm{nonind}(\mathbf{2}, \lambda x.\text{A-ind}(\mathbb{N}, \mathrm{nil_A}))$. Note how $\mathbf{2}$ and $\mathbb{N}$ appear in the code. We will see an example of the slightly more complicated constructor B-ind later.

We will now define $\mathrm{Arg_A}$, which maps a code to the domain of the constructor it represents. $\mathrm{Arg_A}$ will need to take arbitrary $A : \mathrm{Set}$ and $B : A \to \mathrm{Set}$ as parameters to use as $A$ and $B$ in the inductive arguments, since we need $\mathrm{Arg_A}$ to define the $A$ and $B$ we want. We will then have axioms stating that for every code $\gamma : \mathrm{SP}'_{\mathrm{A}}$, there are sets $A_\gamma$, $B_\gamma$ closed under $\mathrm{Arg_A}$, i.e. there is a constructor $\mathrm{intro_A} : \mathrm{Arg_A}(\gamma, A_\gamma, B_\gamma) \to A_\gamma$. $\mathrm{Arg_A}$ has formation rule

$$\frac{\begin{array}{c} A_{\mathrm{ref}}, B_{\mathrm{ref}} : \mathrm{Set} \\ \gamma : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}) \end{array} \quad \begin{array}{c} A : \mathrm{Set} \\ B : A \to \mathrm{Set} \end{array} \quad \begin{array}{c} \mathrm{rep_A} : A_{\mathrm{ref}} \to A \\ \mathrm{rep_{index}} : B_{\mathrm{ref}} \to A \\ \mathrm{rep_B} : (x : B_{\mathrm{ref}}) \to B(\mathrm{rep_{index}}(x)) \end{array}}{\mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \gamma, A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) : \mathrm{Set}}$$

The function $\mathrm{rep_A}$ translates elements in $A_{\mathrm{ref}}$ into the real elements they represent in $A$. Elements in $B_{\mathrm{ref}}$ represent elements $b$ from $B$, but also elements from $A$, as we also need to store the index $a$ such that $b : B(a)$. This index is given by $\mathrm{rep_{index}}$, and $\mathrm{rep_B}$ gives the real element in $B(\mathrm{rep_{index}}(y))$ an element $y : B_{\mathrm{ref}}$ represents.

We are actually only interested in $\mathrm{Arg_A}$ for codes $\gamma : \mathrm{SP'_A}$ for inductive-inductive definitions (i.e. with $A_{\mathrm{ref}} = B_{\mathrm{ref}} = \mathbf{0}$), but we need to consider arbitrary $A_{\mathrm{ref}}$, $B_{\mathrm{ref}}$ for the intermediate codes. For $\gamma : \mathrm{SP'_A}$, we can define a simplified version $\mathrm{Arg'_A} : \mathrm{SP'_A} \to (A : \mathrm{Set}) \to (B : A \to \mathrm{Set}) \to \mathrm{Set}$ by $\mathrm{Arg'_A}(\gamma, A, B) := \mathrm{Arg_A}(\mathbf{0}, \mathbf{0}, \gamma, A, B, !_A, !_A, !_{B \circ !_A})$. (Recall that $!_X : (x : \mathbf{0}) \to X$ is the function given by ex falso quodlibet.)

The definition of $\mathrm{Arg_A}$ also follows the rules for strict positivity in Section 3. We will, for readability, write "_" for arguments which are simply passed on in the recursive call.

The code $\mathrm{nil_A}$ represents the constructor with no argument (i.e. a trivial argument of type $\mathbf{1}$):

$$\mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \mathrm{nil_A}, A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) = \mathbf{1}$$

The code $\mathrm{nonind}(K, \gamma)$ represents one non-inductive argument $k : K$, with the rest of the arguments given by the code $\gamma$ (depending on $k : K$):

$$\begin{aligned}&\mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \mathrm{nonind}(K,\gamma), A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) = \\ &\qquad\qquad\qquad\qquad \big(k : K\big) \times \mathrm{Arg_A}(\_, \_, \gamma(k), \_, \_, \_, \_, \_)\end{aligned}$$

The code $\text{A-ind}(K, \gamma)$ represents one generalised inductive argument $j : K \to A$, with the rest of the arguments given by the code $\gamma$. In the following arguments, $A_{\mathrm{ref}}$ has now been updated to $A_{\mathrm{ref}} + K$, where elements in the old $A_{\mathrm{ref}}$ are mapped to $A$ by the old $\mathrm{rep_A}$, and elements in $K$ are mapped to $A$ by $j$. In effect, this means that we can refer to $j(k)$ for $k : K$ in the following arguments.

$$\begin{aligned}&\mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \text{A-ind}(K,\gamma), A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) = \\ &\qquad\qquad \big(j : K \to A\big) \times \mathrm{Arg_A}(A_{\mathrm{ref}} + K, \_, \gamma, \_, \_, \mathrm{rep_A} \sqcup j, \_, \_)\end{aligned}$$

Finally, the code $\text{B-ind}(K, h_{\mathrm{index}}, \gamma)$ represents one generalised inductive argument $j : (x : K) \to B((\mathrm{rep_A} \circ h_{\mathrm{index}})(x))$, where $\mathrm{rep_A} \circ h_{\mathrm{index}}$ picks out the index of the type of $j(x)$. This time, we can refer to more elements in $B$ afterwards, namely those given by $j$ (and indices given by $\mathrm{rep_A} \circ h_{\mathrm{index}}$):

$$\begin{aligned}&\mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \text{B-ind}(K, h_{\mathrm{index}}, \gamma), A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) = \\ &\quad \big(j : (k : K) \to B((\mathrm{rep_A} \circ h_{\mathrm{index}})(k))\big) \times \\ &\qquad\qquad \mathrm{Arg_A}(\_, B_{\mathrm{ref}} + K, \gamma, \_, \_, \_, \mathrm{rep_{index}} \sqcup (\mathrm{rep_A} \circ h_{\mathrm{index}}), \mathrm{rep_B} \sqcup j)\end{aligned}$$

Let us take a look at the constructor

$$\mathrm{extension} : ((p : \mathrm{Platform}) \times \mathrm{Building}(p)) \to \mathrm{Platform}$$

again. It would have the code $\gamma_{\mathrm{ext}} = \text{A-ind}(\mathbf{1}, \text{B-ind}(\mathbf{1}, \lambda \star . \hat{p}, \mathrm{nil_A})) : \mathrm{SP'_A}$ where $\hat{p} = \mathrm{inr}(\star)$ is the element in $A_{\mathrm{ref}} = \mathbf{0} + \mathbf{1}$ representing the element introduced by A-ind. We have

$$\mathrm{Arg'_A}(\gamma_{\mathrm{ext}}, \mathrm{Platform}, \mathrm{Building}) = (p : \mathbf{1} \to \mathrm{Platform}) \times (\mathbf{1} \to \mathrm{Building}(p(\star))) \times \mathbf{1}$$

which is isomorphic to the domain of extension thanks to the $\eta$-rules for $\mathbf{1}$ (i.e. $X \cong \mathbf{1} \to X$ and $X \cong X \times \mathbf{1}$).

## 4.2 Towards $\mathrm{SP_B}$

If we did not want to use constructors for $A$ as indices for $B$, like for example Building(extension($\langle p, b\rangle$)), we could construct $\mathrm{SP_B}$ in more or less the same way as $\mathrm{SP_A}$ (this corresponds to choosing $k = 0$ below). However, in general we do want to use constructors as indices, hence we have some more work to do. What do we need to know for such a constructor index? We need to know that we want to use a constructor, but that can be encoded in the code. We also need a way to specify the arguments to the constructor, i.e. we need to represent an element of $\mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B)$! If we want to use nested constructors (e.g. extension($\langle$extension($\langle p, b\rangle$)$, b'\rangle$)), we also need to be able to represent elements in $\mathrm{Arg}'_{\mathrm{A}}(\gamma, \mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B_0), B_1)$ etc.

The idea is to represent elements in $\mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B)$ by corresponding elements in "$\mathrm{Arg}'_{\mathrm{A}}(\gamma, A_{\mathrm{ref}}, B_{\mathrm{ref}})$". However, we must first reconstruct the structure of $A_{\mathrm{ref}}$, $B_{\mathrm{ref}}$ as a family, i.e. we will construct $\overline{A_{\mathrm{ref}}} : \mathrm{Set}$, $\overline{B_{\mathrm{ref}}} : \overline{A_{\mathrm{ref}}} \to \mathrm{Set}$, together with functions $\overline{\mathrm{rep_A}} : \overline{A_{\mathrm{ref}}} \to A$ and $\overline{\mathrm{rep_B}} : (x : \overline{A_{\mathrm{ref}}}) \to \overline{B_{\mathrm{ref}}}(x) \to B(\overline{\mathrm{rep_A}}(x))$. Then, we will show that $\overline{\mathrm{rep_A}}$ and $\overline{\mathrm{rep_B}}$ can be lifted to a map $\mathrm{lift}'(\overline{\mathrm{rep_A}}, \overline{\mathrm{rep_B}}) : \mathrm{Arg}'_{\mathrm{A}}(\gamma, \overline{A_{\mathrm{ref}}}, \overline{B_{\mathrm{ref}}}) \to \mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B)$, so that elements in $\mathrm{Arg}'_{\mathrm{A}}(\gamma, \overline{A_{\mathrm{ref}}}, \overline{B_{\mathrm{ref}}})$ indeed can represent elements in $\mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B)$. The process can then be iterated to represent elements in $\mathrm{Arg}'_{\mathrm{A}}(\gamma, \mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B_0), B_1)$ etc.

$\overline{A_{\mathrm{ref}}}$ should consist of representatives $\overline{a}$ for elements $a$ in $A$, and $\overline{B_{\mathrm{ref}}}(\overline{a})$ should consist of representatives for elements in $B(a)$. The representative $\overline{a}$ could either be from $A_{\mathrm{ref}}$ (with $a = \mathrm{rep_A}(\overline{a})$), in which case we do not know any elements in $B(a)$, or from $B_{\mathrm{ref}}$ (with $a = \mathrm{rep_{index}}(\overline{a})$), in which case we know a single element in $B(a)$, namely $\mathrm{rep_B}(\overline{a})$. Therefore, for $A_{\mathrm{ref}}, B_{\mathrm{ref}} : \mathrm{Set}$, we define

$$\overline{A_{\mathrm{ref}}} \coloneqq A_{\mathrm{ref}} + B_{\mathrm{ref}} \ , \qquad \overline{B_{\mathrm{ref}}} \coloneqq (\lambda x.\mathbf{0}) \sqcup (\lambda x.\mathbf{1}) \ ,$$

i.e. $\overline{B_{\mathrm{ref}}}(\mathrm{inl}(a)) = \mathbf{0}$ and $\overline{B_{\mathrm{ref}}}(\mathrm{inr}(b)) = \mathbf{1}$.

Mapping representatives in $\overline{A_{\mathrm{ref}}}$ to the elements they represent in $A$ is now easy: we map $\mathrm{inl}(\overline{a}) : \overline{A_{\mathrm{ref}}}$ to $\mathrm{rep_A}(\overline{a})$ and $\mathrm{inr}(\overline{a})$ to $\mathrm{rep_{index}}(\overline{a})$. For $\overline{B_{\mathrm{ref}}}$, we want to map representatives in $\overline{B_{\mathrm{ref}}}(\overline{a})$ to the elements they represent in $B(a)$. However, we only have to consider $\overline{B_{\mathrm{ref}}}(\mathrm{inr}(x)) = \mathbf{1}$, as there are no elements in $\overline{B_{\mathrm{ref}}}(\mathrm{inl}(x)) = \mathbf{0}$. We map $\star : \overline{B_{\mathrm{ref}}}(\mathrm{inr}(\overline{a}))$ to $\mathrm{rep_B}(\overline{a})$. To sum up, we can define maps $\overline{\mathrm{rep_A}} : \overline{A_{\mathrm{ref}}} \to A$, $\overline{\mathrm{rep_B}} : (x : \overline{A_{\mathrm{ref}}}) \to \overline{B_{\mathrm{ref}}}(x) \to B(\overline{\mathrm{rep_A}}(x))$ by $\overline{\mathrm{rep_A}} \coloneqq \mathrm{rep_A} \sqcup \mathrm{rep_{index}}$ and $\overline{\mathrm{rep_B}} \coloneqq (\lambda x.!_{B\circ !_A}) \sqcup (\lambda x \star .\mathrm{rep_B}(x))$.

We now want to lift these maps to a map $\mathrm{rep_{A,1}} : \mathrm{Arg}'_{\mathrm{A}}(\gamma, \overline{A_{\mathrm{ref}}}, \overline{B_{\mathrm{ref}}}) \to \mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B)$. This is made possible by the following more general result for arbitrary families $A$, $B$, $A^*$, $B^*$ with respective representing functions $\mathrm{rep_A}$, $\mathrm{rep}^*_{\mathrm{A}}$ etc. Assume we have maps $g : A \to A^*$, $g' : (x : A) \to B(x) \to B^*(g(x))$ that respect the translations $\mathrm{rep_A}$ and $\mathrm{rep}^*_{\mathrm{A}}$, i.e. we have a proof $p$ that $g(\mathrm{rep_A}(x)) = \mathrm{rep}^*_{\mathrm{A}}(x)$ for all $x : A_{\mathrm{ref}}$. Then we can lift $g$ to a map

$$\begin{aligned}\mathrm{lift}(g, g', p) : \mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \gamma, A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) \to \\ \mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \gamma, A^*, B^*, \mathrm{rep}^*_{\mathrm{A}}, \mathrm{rep}^*_{\mathrm{index}}, \mathrm{rep}^*_{\mathrm{B}}) \ .\end{aligned}$$

This can be done component-wise by using the translation functions $g$, $g'$, and using the proof that $g(\mathrm{rep_A}(x)) = \mathrm{rep}^*_{\mathrm{A}}(x)$ to go from $B^*(g(\mathrm{rep_A}(x)))$

to $B^*(\mathrm{rep}^*_{\mathrm{A}}(x))$ in the B-ind case. (It might be worth pointing out that this also works in intensional type theory, as we only need that $g \circ \mathrm{rep}_{\mathrm{A}}$ and $\mathrm{rep}^*_{\mathrm{A}}$ are pointwise equal.) We will omit the definition here for lack of space (see [14] for details). Instead, recall the code $\gamma_{\mathrm{ext}}$ for the constructor

$$\mathrm{extension} : ((p : \mathrm{Platform}) \times \mathrm{Building}(p)) \to \mathrm{Platform} \enspace .$$

An element from $\mathrm{Arg}'_{\mathrm{A}}(\gamma_{\mathrm{ext}}, \mathrm{Platform}, \mathrm{Building})$ is of the form[3] $z = \langle p, b, \star \rangle$ where $p : \mathrm{Platform}$ and $b : \mathrm{Building}(p)$. Given functions $g : \mathrm{Platform} \to A^*$ and $g' : (x : \mathrm{Platform}) \to \mathrm{Building}(x) \to B^*(g(x))$ for some other $A^*$, $B^*$, then $z$ would be mapped to

$$\mathrm{lift}(g, g', \mathrm{triv})(z) = \langle g(p), g'(p, b), \star \rangle : \mathrm{Arg}'_{\mathrm{A}}(\gamma_{\mathrm{ext}}, A^*, B^*) \enspace .$$

Here, triv is a trivial proof that $g(\mathrm{rep}_{\mathrm{A}}(x)) = \mathrm{rep}^*_{\mathrm{A}}(x)$ for every $x : \mathbf{0}$. This will be a valid proof for every $\gamma : \mathrm{SP}'_{\mathrm{A}}$, so once again we define a simplified version

$$\mathrm{lift}'(g, g') : \mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B) \to \mathrm{Arg}'_{\mathrm{A}}(\gamma, A^*, B^*)$$

by $\mathrm{lift}'(g, g') \coloneqq \mathrm{lift}(g, g', \mathrm{triv})$.

In our specific case, let for $\gamma : \mathrm{SP}'_{\mathrm{A}}$, $A_{\mathrm{ref}}, B_{\mathrm{ref}} : \mathrm{Set}$ and $\mathrm{rep}_{\mathrm{A}} : A_{\mathrm{ref}} \to A$, $\mathrm{rep}_{\mathrm{index}} : B_{\mathrm{ref}} \to A$, $\mathrm{rep}_{\mathrm{B}} : (b : B_{\mathrm{ref}}) \to B(\mathrm{rep}_{\mathrm{index}}(b))$

$$\begin{aligned}
\overline{\mathrm{arg}_{\mathrm{A}}}(\gamma, A_{\mathrm{ref}}, B_{\mathrm{ref}}) &\coloneqq \mathrm{Arg}'_{\mathrm{A}}(\gamma, \overline{A_{\mathrm{ref}}}, \overline{B_{\mathrm{ref}}}) \enspace , \\
\overline{\mathrm{lift}}(\mathrm{rep}_{\mathrm{A}}, \mathrm{rep}_{\mathrm{index}}, \mathrm{rep}_{\mathrm{B}}) &\coloneqq \mathrm{lift}'(\overline{\mathrm{rep}_{\mathrm{A}}}, \overline{\mathrm{rep}_{\mathrm{B}}}) : \overline{\mathrm{arg}_{\mathrm{A}}}(\gamma, A_{\mathrm{ref}}, B_{\mathrm{ref}}) \to \mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B) \, .
\end{aligned}$$

We can now represent elements in $\mathrm{Arg}'_{\mathrm{A}}(\gamma, A, B)$ by elements in $\overline{\mathrm{arg}_{\mathrm{A}}}(\gamma, A_{\mathrm{ref}}, B_{\mathrm{ref}})$ via $\mathrm{rep}_{\mathrm{A},1} \coloneqq \overline{\mathrm{lift}}(\mathrm{rep}_{\mathrm{A}}, \mathrm{rep}_{\mathrm{index}}, \mathrm{rep}_{\mathrm{B}})$. For example, consider $\gamma_{\mathrm{ext}}$ once again. Suppose that we want to specify an element of $\mathrm{Arg}'_{\mathrm{A}}(\gamma_{\mathrm{ext}}, \mathrm{Platform}, \mathrm{Building})$, i.e. an argument to the extension constructor, and we have $A_{\mathrm{ref}} = B_{\mathrm{ref}} = \mathbf{0} + \mathbf{1}$. $\overline{A_{\mathrm{ref}}}$ then consists of two elements, namely $\hat{p} = \mathrm{inl}(\mathrm{inr}(\star))$ and $\widehat{pb} = \mathrm{inr}(\mathrm{inr}(\star))$. We have that $\overline{B_{\mathrm{ref}}}(\hat{p}) = \mathbf{0}$ and $\overline{B_{\mathrm{ref}}}(\widehat{pb}) = \mathbf{1}$. In other words, there is only one element $\widehat{\langle pb \rangle} = \langle \widehat{pb}, \star, \star \rangle$ of $\mathrm{Arg}'_{\mathrm{A}}(\gamma_{\mathrm{ext}}, \overline{A_{\mathrm{ref}}}, \overline{B_{\mathrm{ref}}})$, and $\mathrm{rep}_{\mathrm{A},1}(\widehat{\langle pb \rangle}) = \langle \mathrm{rep}_{\mathrm{index}}(\widehat{pb}), \mathrm{rep}_{\mathrm{B}}(\widehat{pb}), \star \rangle = \langle p, b, \star \rangle$ where $\mathrm{rep}_{\mathrm{index}}(\widehat{pb}) = p : \mathrm{Platform}$, $\mathrm{rep}_{\mathrm{B}}(\widehat{pb}) = b : \mathrm{Building}(p)$.

Let us now generalise this to multiple nestings of constructors. Given a sequence $\vec{B}_{\mathrm{ref}(n)} = B_{\mathrm{ref},\,0}, B_{\mathrm{ref},\,1}, \ldots, B_{\mathrm{ref},\,n-1}$ of sets, we iterate $\overline{\mathrm{arg}_{\mathrm{A}}}$ by defining

$$\begin{aligned}
\mathrm{arg}^0_{\mathrm{A}}(\gamma, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}(0)}) &= A_{\mathrm{ref}} \enspace , \\
\mathrm{arg}^{n+1}_{\mathrm{A}}(\gamma, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}(n+1)}) &= \overline{\mathrm{arg}_{\mathrm{A}}}(\gamma, \sum_{i=0}^{n} \mathrm{arg}^i_{\mathrm{A}}(\gamma, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}(i)}), B_{\mathrm{ref},\,n}) \enspace .
\end{aligned}$$

$\mathrm{arg}^k_{\mathrm{A}}$ should represent $k$ nested constructors. The corresponding iteration of $\mathrm{Arg}'_{\mathrm{A}}$ justifying this, is

$$\begin{aligned}
\mathrm{Arg}^0_{\mathrm{A}}(\gamma, A, \vec{B}_{(0)}) &= A \enspace , \\
\mathrm{Arg}^{n+1}_{\mathrm{A}}(\gamma, A, \vec{B}_{(n+1)}) &= \mathrm{Arg}'_{\mathrm{A}}(\gamma, \sum_{i=0}^{n} \mathrm{Arg}^i_{\mathrm{A}}(\gamma, A, \vec{B}_{(i)}), \bigsqcup_{i=0}^{n} B_i) \enspace ,
\end{aligned}$$

[3] We identify $\mathbf{1} \to X$ and $X$ for the sake of readability. If not, $p$ would be $\lambda \star . p$ and so on.

where $\vec{B}_{(n)}$ is a sequence $B_0, B_1, \ldots, B_{n-1}$ of families $B_i : \mathrm{Arg}^i_{\mathrm{A}}(\gamma_A, A, \vec{B}_{(i-1)}) \to$ Set.

Now assume that we have a sequence $\vec{B}_{\mathrm{ref}(n)}$ of sets and a sequence $\vec{B}_{(n)}$ of families of sets as above. If we now in addition to $\mathrm{rep}_{\mathrm{A}} : A_{\mathrm{ref}} \to A$, also have functions $\mathrm{rep}_{\mathrm{index},i} : B_{\mathrm{ref},\ i} \to \mathrm{Arg}^i_{\mathrm{A}}(\gamma, A, \vec{B})$, and $\mathrm{rep}_{\mathrm{B},i} : (x : B_{\mathrm{ref},\ i}) \to B_i(\mathrm{rep}_{\mathrm{index},i}(x))$ (we will assume all this when working with $\mathrm{SP}_{\mathrm{B}}$), we can now construct functions $\mathrm{rep}_{\mathrm{A},n} : \mathrm{arg}^n_{\mathrm{A}}(\gamma, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}) \to \mathrm{Arg}^n_{\mathrm{A}}(\gamma, A, \vec{B})$ with the help of $\mathrm{lift}'$ by defining

$$\begin{aligned} \mathrm{rep}_{\mathrm{A},0} &= \mathrm{rep}_{\mathrm{A}} , \\ \mathrm{rep}_{\mathrm{A},n+1} &= \overline{\mathrm{lift}}\Big(\coprod_{i=0}^{n} \mathrm{rep}_{\mathrm{A},i}, \mathrm{in}_n \circ \mathrm{rep}_{\mathrm{index},n}, \mathrm{rep}_{\mathrm{B},n}\Big) . \end{aligned}$$

Note that $\mathrm{rep}_{\mathrm{A},1}$ as defined earlier is an instance of this definition.

## 4.3 $\mathbf{SP_B}$, $\mathbf{Arg_B}$ and $\mathbf{Index_B}$

The datatype $\mathrm{SP}_{\mathrm{B}}$ of codes for constructors for $B$ is just as $\mathrm{SP}_{\mathrm{A}}$, but with two differences: first, we can refer to constructors of $A$ (so we will need a code $\gamma_{\mathrm{A}} : \mathrm{SP}'_{\mathrm{A}}$ to know their form, and sets $B_{\mathrm{ref},\ 0}, B_{\mathrm{ref},\ 1}, \ldots B_{\mathrm{ref},\ i}, \ldots$ to represent elements in $B$ indexed by $i$ nested constructors). Second, we also need to specify an index for the codomain of the constructor (so we will store this index in the $\mathrm{nil}_{\mathrm{B}}$ code).

All constructions from now on will be parameterised on the maximum number $k$ of nested constructors for $A$ that we are using, so we are really introducing $\mathrm{SP}_{\mathrm{B},k}$, $\mathrm{Arg}_{\mathrm{B},k}$ etc. However, we will work with an arbitrary $k$ but suppress it as a premise. With this in mind, we have formation rule

$$\frac{\gamma_A : \mathrm{SP}'_{\mathrm{A}} \quad A_{\mathrm{ref}} : \mathrm{Set} \quad B_{\mathrm{ref},\ 0}, B_{\mathrm{ref},\ 1}, \ldots, B_{\mathrm{ref},\ k} : \mathrm{Set}}{\mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, B_{\mathrm{ref},\ 1}, \ldots, B_{\mathrm{ref},\ k}) : \mathrm{Type}}$$

Let $\mathrm{SP}'_{\mathrm{B}}(\gamma_A) \coloneqq \mathrm{SP}_{\mathrm{B}}(\gamma_A, \mathbf{0}, \mathbf{0}, \ldots, \mathbf{0})$ for $\gamma_A : \mathrm{SP}'_{\mathrm{A}}$ be the code of inductive-inductive definitions.

The introduction rules for $\mathrm{SP}_{\mathrm{B}}$ are very similar to the rules for $\mathrm{SP}_{\mathrm{A}}$, but now we specify an index in $\mathrm{nil}_{\mathrm{B}}$, and we have $k+1$ rules $\mathrm{B}_0$-ind$, \ldots,$ $\mathrm{B}_k$-ind corresponding to how many nested constructors for $A$ we want to use:

$$\frac{a_{\mathrm{index}} : +_{i=0}^{k} \mathrm{arg}^i_{\mathrm{A}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}})}{\mathrm{nil}_{\mathrm{B}}(a_{\mathrm{index}}) : \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ k})}$$

$$\frac{K : \mathrm{Set} \quad \gamma : K \to \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ k})}{\mathrm{nonind}(K, \gamma) : \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ k})}$$

$$\frac{K : \mathrm{Set} \quad \gamma : \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}} + K, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ k})}{\mathrm{A\text{-}ind}(K, \gamma) : \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ k})}$$

$$\frac{\begin{array}{c} h_{\mathrm{index}} : K \to \mathrm{arg}^{\ell}_{\mathrm{A}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}) \\ K : \mathrm{Set} \quad \gamma : \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ \ell} + K, \ldots, B_{\mathrm{ref},\ k}) \end{array}}{\mathrm{B}_{\ell}\text{-ind}(K, h_{\mathrm{index}}, \gamma) : \mathrm{SP}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, B_{\mathrm{ref},\ 0}, \ldots, B_{\mathrm{ref},\ k})}$$

The rules $\mathrm{nil}_{\mathrm{B}}$, nonind and A-ind have the same meaning as before, but B-ind has been split up into several rules. The code $\mathrm{B}_\ell\text{-ind}(K, h_{\mathrm{index}}, \gamma)$ represents an inductive argument of type $(x : K) \to B(i_\ell(x))$ with the index $i_\ell(x)$, using $\ell$ nested constructors for $A$, given by $h_{\mathrm{index}}$. Hence $h_{\mathrm{index}}$ has codomain $\mathrm{arg}^\ell_{\mathrm{A}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}})$. For this to work, we will need $k+1$ families $B_i : \mathrm{Arg}^i_{\mathrm{A}}(\gamma_A, A, \vec{B}_{(i-1)}) \to \mathrm{Set}$ to interpret these nested constructor indices, together with functions $\mathrm{rep}_{\mathrm{index},i} : B_{\mathrm{ref},\, i} \to \mathrm{Arg}^i_{\mathrm{A}}(\gamma, A, \vec{B}_{(i-1)})$ and $\mathrm{rep}_{\mathrm{B},i} : (x : B_{\mathrm{ref},\, i}) \to B_i(\mathrm{rep}_{\mathrm{index},i}(x))$ to map elements to the real elements they represent. Recall that from this and $\mathrm{rep}_{\mathrm{A}} : A_{\mathrm{ref}} \to A$, we can construct functions $\mathrm{rep}_{\mathrm{A},\ell} : \mathrm{arg}^\ell_{\mathrm{A}}(\gamma, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}) \to \mathrm{Arg}^\ell_{\mathrm{A}}(\gamma, A, \vec{B})$.

Every case of $\mathrm{Arg}_{\mathrm{B}}$ is the same as the corresponding case for $\mathrm{Arg}_{\mathrm{A}}$, except for $\mathrm{B}_\ell$-ind, where $B$ has been replaced by $B_\ell$ and $\mathrm{rep}_{\mathrm{A}}$ by $\mathrm{rep}_{\mathrm{A},\ell}$ (we write "_" for passed on arguments and "__" for passed on sequents of arguments in the recursive call):

$$\mathrm{Arg}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \mathrm{nil}_{\mathrm{B}}(a_{\mathrm{index}}), A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}) = \mathbf{1}$$
$$\mathrm{Arg}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \mathrm{nonind}(K,\gamma), A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}) = \big(k : K\big) \times \mathrm{Arg}_{\mathrm{B}}(\_, \_, \_\_, \gamma(k), \_, \_\_, \_, \_\_, \_\_)$$
$$\mathrm{Arg}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \text{A-ind}(K,\gamma), A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}) = \big(j : K \to A\big) \times \mathrm{Arg}_{\mathrm{B}}(\_, A_{\mathrm{ref}} + K, \_\_, \gamma, \_, \_\_, \mathrm{rep}_{\mathrm{A}} \sqcup j, \_\_, \_\_)$$
$$\begin{aligned}&\mathrm{Arg}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \mathrm{B}_\ell\text{-ind}(K, h_{\mathrm{index}}, \gamma), A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}) = \\ &\big(j : (k : K) \to B_\ell((\mathrm{rep}_{\mathrm{A},\ell} \circ h_{\mathrm{index}})(k))\big) \times \\ &\mathrm{Arg}_{\mathrm{B}}(\_, \_, \_\_, B_{\mathrm{ref},\,\ell}{+}K, \_\_, \gamma, \_, \_\_, \_, \_\_, \mathrm{rep}_{\mathrm{index},\ell}{\sqcup}(\mathrm{rep}_{\mathrm{A},\ell}{\circ}h_{\mathrm{index}}), \_\_, \_\_, \mathrm{rep}_{\mathrm{B},\ell}{\sqcup}j, \_\_)\end{aligned}$$

The last missing piece is now $\mathrm{Index}_{\mathrm{B}}$, which to each $b : \mathrm{Arg}'_{\mathrm{B}}(\gamma_A, \gamma_B, A, \vec{B})$ assigns the index $a$ such that the element constructed from $b$ is in $B(a)$. With $\gamma_A$, $A_{\mathrm{ref}}$, $B_{\mathrm{ref},\, i}$ etc as above, $\mathrm{Index}_{\mathrm{B}}$ has formation rule

$$\begin{aligned}&\mathrm{Index}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \gamma_B, A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}) : \\ &\quad \mathrm{Arg}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \gamma_B, A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}) \to \mathop{+}_{i=0}^{k} \mathrm{Arg}^i_{\mathrm{A}}(\gamma_A, A, \vec{B}).\end{aligned}$$

$\mathrm{Index}_{\mathrm{B}}$ will take $a_{\mathrm{index}} : +_{i=0}^{k} \mathrm{arg}^i_{\mathrm{A}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}})$ which is stored in $\mathrm{nil}_{\mathrm{B}}$ and map it to the index element it represents in $+_{i=0}^{k} \mathrm{Arg}^i_{\mathrm{A}}(\gamma_A, A, \vec{B})$. For other codes, it will follow exactly the same pattern as $\mathrm{Arg}_{\mathrm{B}}$, so we omit the rules for them here.

$$\mathrm{Index}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \mathrm{nil}_{\mathrm{B}}(a_{\mathrm{index}}), A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}, \star) = (\coprod_{i=0}^{k} \mathrm{rep}_{\mathrm{A},i})(a_{\mathrm{index}})$$
$$\mathrm{Index}_{\mathrm{B}}(\gamma_A, A_{\mathrm{ref}}, \vec{B}_{\mathrm{ref}}, \mathrm{nonind}(K,\gamma), A, \vec{B}, \mathrm{rep}_{\mathrm{A}}, \vec{\mathrm{rep}}_{\mathrm{index}}, \vec{\mathrm{rep}}_{\mathrm{B}}, \langle k, y\rangle) = \mathrm{Index}_{\mathrm{B}}(\_, \_, \_\_, \gamma(k), \_, \_\_, \_, \_\_, \_\_, y)$$
$$\vdots$$

For codes $\gamma_B : \mathrm{SP}'_{\mathrm{B}}(\gamma_A)$ for inductive-inductive definitions, let

$$\begin{aligned}\mathrm{Arg}'_{\mathrm{B}}(\gamma_A, \gamma_B, A, \vec{B}) &\coloneqq \mathrm{Arg}_{\mathrm{B}}(\gamma_A, \mathbf{0}, \vec{\mathbf{0}}, \gamma_B, A, \vec{B}, !_A, \vec{!}_{\mathrm{Arg}_{\mathrm{A}}}, \vec{!}_{B\circ !}) \ , \\ \mathrm{Index}'_{\mathrm{B}}(\gamma_A, \gamma_B, A, \vec{B}) &\coloneqq \mathrm{Index}_{\mathrm{B}}(\gamma_A, \mathbf{0}, \vec{\mathbf{0}}, \gamma_B, A, \vec{B}, !_A, !_{\mathrm{Arg}_{\mathrm{A}}}, !_{B\circ !}) \ .\end{aligned}$$

As an illustration, let us consider the constructor

$$\mathrm{hangingUnder} : ((p : \mathrm{Platform}) \times (b : \mathrm{Building}(p))) \to \mathrm{Building}(\mathrm{extension}(\langle p, b\rangle)).$$

Here it is interesting to see that the index of the codomain of the constructor uses the constructor extension, so it will be represented by an element from $\overline{\mathrm{arg}_{\mathrm{A}}}(\gamma_{\mathrm{ext}}, A_{\mathrm{ref}}, B_{\mathrm{ref}})$. We end up with the code $\gamma_{\mathrm{hu}} = \text{A-ind}(\mathbf{1}, \mathrm{B}_0\text{-ind}(\mathbf{1}, \lambda \star .\hat{p}, \mathrm{nil}_{\mathrm{B}}(\mathrm{in}_1(\widehat{\langle pb\rangle}))))$, where $\hat{p} = \mathrm{inr}(\star)$ and $\widehat{\langle pb\rangle} = \langle \mathrm{inr}(\mathrm{inr}(\star)), \star, \star\rangle$. (The reader is invited to check that $\widehat{\langle pb\rangle} : \overline{\mathrm{arg}_{\mathrm{A}}}(\gamma_{\mathrm{ext}}, A_{\mathrm{ref}}, B_{\mathrm{ref}})$ at that point in the construction of $\gamma_{\mathrm{hu}}$.) We get

$$\mathrm{Arg}'_{\mathrm{B}}(\gamma_{\mathrm{ext}}, \gamma_{\mathrm{hu}}, P, B) = (p : \mathbf{1} \to P) \times (b : \mathbf{1} \to B(p(\star))) \times \mathbf{1}$$

and $\mathrm{Index}'_{\mathrm{B}}(\gamma_{\mathrm{ext}}, \gamma_{\mathrm{hu}}, P, B, \langle p, b, \star\rangle) = \mathrm{in}_1(\langle p, b, \star\rangle)$.

## 4.4 Formation and Introduction Rules

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

Formation rules:

$$A_{\gamma_A,\gamma_B} : \mathrm{Set} \ , \qquad B_{\gamma_A,\gamma_B} : A_{\gamma_A,\gamma_B} \to \mathrm{Set} \ .$$

Introduction rule for $A_{\gamma_A,\gamma_B}$:

$$\frac{a : \mathrm{Arg}'_{\mathrm{A}}(\gamma_A, A_{\gamma_A,\gamma_B}, B_{\gamma_A,\gamma_B})}{\mathrm{intro}_{\mathrm{A}}(a) : A_{\gamma_A,\gamma_B}}$$

For the introduction rule for $B_{\gamma_A,\gamma_B}$, we need some preliminary definitions. We have $B_{\gamma_A,\gamma_B} : A_{\gamma_A,\gamma_B} \to \mathrm{Set}$, but need $B_i : \mathrm{Arg}^i_{\mathrm{A}}(\gamma_A, A_{\gamma_A,\gamma_B}, \vec{B}_{(i-1)}) \to \mathrm{Set}$ for $0 \le i \le k$ to give to $\mathrm{Arg}'_{\mathrm{B}}$. We can assemble such $B_i$'s from $B_{\gamma_A,\gamma_B}$, $\mathrm{intro}_{\mathrm{A}}$ and $\mathrm{lift}'$. To do so, define in a step by step manner

$$\begin{aligned}\mathrm{intro}_n : \mathrm{Arg}^n_{\mathrm{A}}(\gamma_A, A_{\gamma_A,\gamma_B}, B_0, \ldots, B_{n-1}) &\to A_{\gamma_A,\gamma_B} \\ B_n : \mathrm{Arg}^n_{\mathrm{A}}(\gamma_A, A_{\gamma_A,\gamma_B}, B_0, \ldots, B_{n-1}) &\to \mathrm{Set}\end{aligned}$$

(i.e. introduce first $\mathrm{intro}_0$, $B_0$, then $\mathrm{intro}_1$, $B_1$ and so on) by

$$\begin{aligned}\mathrm{intro}_0 &= \mathrm{id} \\ \mathrm{intro}_{n+1} &= \mathrm{intro}_{\mathrm{A}} \circ \mathrm{lift}'(\bigsqcup_{i=0}^{n} \mathrm{intro}_i, \bigsqcup_{i=0}^{n} (\lambda a.\mathrm{id})) \\ B_i(x) &= B_{\gamma_A,\gamma_B}(\mathrm{intro}_i(x))\end{aligned}$$

Hence the introduction rule for $B_{\gamma_A,\gamma_B}$ can be given as:

$$\frac{b : \mathrm{Arg}'_{\mathrm{B}}(\gamma_A, A_{\gamma_A,\gamma_B}, B_{\gamma_A,\gamma_B}, B_1, \ldots, B_k)}{\mathrm{intro}_{\mathrm{B}}(b) : B_{\gamma_A,\gamma_B}(\overline{\mathrm{index}}(b))}$$

where $\overline{\mathrm{index}}$ takes the index in $+_{i=1}^{k} \mathrm{Arg}^i_{\mathrm{A}}(\gamma_A, A_{\gamma_A,\gamma_B}, \vec{B})$ returned by $\mathrm{Index}'_{\mathrm{B}}$ and applies the right $\mathrm{intro}_i$ to it, i.e.

$$\overline{\mathrm{index}} = (\bigsqcup_{i=0}^{k} \mathrm{intro}_i) \circ \mathrm{Index}'_{\mathrm{B}}(\gamma_A, \gamma_B, A_{\gamma_A,\gamma_B}, B_0, \ldots, B_k) \ .$$

Elimination rules similar to the rules for indexed inductive definitions can also be formulated, but we here omit them due to lack of space (see [14] for full details).

### 4.5 Contexts and Types Again

As a final example, let us construct Ctxt and Ty from Section 1. With the abbreviation $\gamma_0 +_{SP} \gamma_1 \coloneqq \mathrm{nonind}(\mathbf{2}, \lambda x.\mathbf{if}\ x\ \mathbf{then}\ \gamma_0\ \mathbf{else}\ \gamma_1)$, we can encode several constructors into one. The codes for the contexts and types are

$$\begin{aligned}
\gamma_{\mathrm{Ctxt}} &= \mathrm{nil}_{\mathrm{A}} +_{SP} \text{A-ind}(\mathbf{1}, \text{B-ind}(\mathbf{1}, \lambda \star .\mathrm{inr}(\star), \mathrm{nil}_{\mathrm{A}})) : \mathrm{SP}'_{\mathrm{A}} \\
\gamma_{\text{'set'}} &= \text{A-ind}(\mathbf{1}, \mathrm{nil}_{\mathrm{B}}(\mathrm{inl}(\mathrm{inr}(\star)))) \\
\gamma_{\Pi} &= \text{A-ind}(\mathbf{1}, \mathrm{B}_0\text{-ind}(\mathbf{1}, \lambda \star .\mathrm{inr}(\star), \mathrm{B}_1\text{-ind}(\mathbf{1}, \lambda \star .\langle \mathrm{ff}, \langle \lambda \star .\mathrm{inr}(\mathrm{inr}(\star)), \\
&\qquad \langle \lambda \star . \star, \star \rangle\rangle\rangle, \mathrm{nil}_{\mathrm{B}}(\mathrm{inl}(\mathrm{inr}(\star)))))) \\
\gamma_{\mathrm{Ty}} &= \gamma_{\text{'set'}} +_{SP} \gamma_{\Pi} : \mathrm{SP}'_{\mathrm{B}}(\gamma_{\mathrm{Ctxt}}) \ .
\end{aligned}$$

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

$$\begin{aligned}
&\varepsilon : \mathrm{Ctxt} && \text{'set'} : (\Gamma : \mathrm{Ctxt}) \to \mathrm{Ty}(\Gamma) \\
&\varepsilon = \mathrm{intro}_{\mathrm{A}}\langle \mathrm{tt}, \star \rangle \ , && \text{'set'}(\Gamma) = \mathrm{intro}_{\mathrm{B}}\langle \mathrm{tt}, \langle (\lambda \star .\Gamma), \star \rangle\rangle \ ,
\end{aligned}$$

$$\begin{aligned}
&\mathrm{cons} : (\Gamma : \mathrm{Ctxt}) \to \mathrm{Ty}(\Gamma) \to \mathrm{Ctxt} \\
&\mathrm{cons}(\Gamma, b) = \mathrm{intro}_{\mathrm{A}}(\langle \mathrm{ff}, \langle (\lambda \star .\Gamma), \langle (\lambda \star .b), \star \rangle\rangle\rangle) \quad ,
\end{aligned}$$

$$\begin{aligned}
&\Pi : (\Gamma : \mathrm{Ctxt}) \to (A : \mathrm{Ty}(\Gamma)) \to \mathrm{Ty}(\mathrm{cons}(\Gamma, A)) \to \mathrm{Ty}(\Gamma) \\
&\Pi(\Gamma, A, B) = \mathrm{intro}_{\mathrm{B}}(\langle \mathrm{ff}, \langle (\lambda \star .\Gamma), \langle (\lambda \star .A), \langle (\lambda \star .B), \star \rangle\rangle\rangle\rangle) \ .
\end{aligned}$$

## 5 A Set-Theoretic Model

Even though $\mathrm{SP}_{\mathrm{A}}$ and $\mathrm{SP}_{\mathrm{B}}$ are straightforward (large) inductive definitions, this axiomatisation does not reduce inductive-inductive definitions to indexed inductive definitions, since the formation and introduction rules are not instances of ordinary indexed inductive definitions. (However, we do believe that induction-induction *can* be reduced to indexed induction with a bit of more work, and plan to publish an article about this in the future.) To make sure that our theory is consistent, it is thus neccessary to construct a model.

A model of our theory can be constructed in ZFC set theory, extended by two strongly inaccessible cardinals $\mathfrak{i}_0 < \mathfrak{i}_1$ in order to interpret Set and Type.

Our model will be a simpler version of the models developed in [9,11]. Here we present the main ideas; more details can be found in [14]. See Aczel [1] for a more detailed treatment of interpreting type theory in set theory.

For every expression $A$ of our type theory, we will give an interpretation $[\![A]\!]_\rho$, which might be undefined. Open terms will be interpreted relative to an environment $\rho$, i.e. a function mapping variables to terms, and contexts will be interpreted as sets of environments.

We interpret the logical framework exactly as in [9]; Each type is interpreted as a set, $a : A$ is interpreted as $a \in A$, $(x : A) \to B$ as the set-theoretic Cartesian product $\Pi_{x \in A} B$ etc. Set is interpreted as $V_{\mathfrak{i}_0}$ and Type as $V_{\mathfrak{i}_1}$.

$\mathrm{SP_A}$ can be interpreted as the least set $[\![\mathrm{SP_A}]\!](D, D')$ such that

$$[\![\mathrm{SP_A}]\!](D, D') = 1 + \sum_{K \in [\![\mathrm{Set}]\!]} (K \to [\![\mathrm{SP_A}]\!](D, D')) + \sum_{K \in [\![\mathrm{Set}]\!]} [\![\mathrm{SP_A}]\!](D + K, D')$$
$$+ \sum_{K \in [\![\mathrm{Set}]\!]} \sum_{h : K \to D} [\![\mathrm{SP_A}]\!](D, D' + K) \ ,$$

which must be an element of $[\![\mathrm{Type}]\!] = V_{\mathfrak{i}_1}$ by the inaccessibility of $\mathfrak{i}_1$. We define

$$[\![\mathrm{nil_A}]\!] :\simeq \langle 0, 0 \rangle \ , \qquad [\![\text{A-ind}(K, \gamma)]\!] :\simeq \langle 2, \langle K, \gamma \rangle \rangle \ ,$$
$$[\![\mathrm{nonind}(K, \gamma)]\!] := \langle 1, \langle K, \gamma \rangle \rangle \ , \qquad [\![\text{B-ind}(K, h, \gamma)]\!] :\simeq \langle 3, \langle K, \langle h, \gamma \rangle \rangle \rangle \ .$$

$[\![\mathrm{SP_B}]\!]$ and $[\![\mathrm{nil_B}]\!]$, $[\![\mathrm{B}_\ell\text{-ind}]\!]$ are defined analogously. $[\![\mathrm{Arg_A}]\!]$. $[\![\mathrm{Arg_B}]\!]$, and $[\![\mathrm{Index_B}]\!]$ are defined according to their equations.

Finally, we have to interpret $A_{\gamma_A, \gamma_B}$, $B_{\gamma_A, \gamma_B}$, $\mathrm{intro_A}$ and $\mathrm{intro_B}$. Let

$$[\![A_{\gamma_A, \gamma_B}]\!] :\simeq A^{\mathfrak{i}_0} \ , \quad [\![B_{\gamma_A, \gamma_B}]\!](a) :\simeq B^{\mathfrak{i}_0}(a) \ , \quad [\![\mathrm{intro_A}]\!](a) :\simeq a \ , \quad [\![\mathrm{intro_B}]\!](b) :\simeq b \ ,$$

where $A^\alpha$ and $B^\alpha$ are simultaneously defined by recursion on $\alpha$ as

$$A^\alpha := [\![\mathrm{Arg'_A}]\!](\gamma_A, A^{<\alpha}, B^{<\alpha}) \ ,$$
$$B^\alpha(a) := \{ b \mid b \in [\![\mathrm{Arg'_B}]\!](\gamma_A, \gamma_B, A^{<\alpha}, \vec{B^{<\alpha}}) \wedge [\![\mathrm{Index'_B}]\!](\gamma_A, \gamma_B, A^{<\alpha}, \vec{B^{<\alpha}}, b) = a \} \ .$$

**Theorem 1 (Soundness)**

*(i)* *If* $\vdash \Gamma$ *context, then* $[\![\Gamma]\!] \downarrow$.

*(ii)* *If* $\Gamma \vdash A : E$, *then* $[\![\Gamma]\!] \downarrow$, *and for all* $\rho \in [\![\Gamma]\!]$, $[\![A]\!]_\rho \in [\![E]\!]_\rho$, *and also* $[\![E]\!]_\rho \in [\![\mathit{Type}]\!]$ *if* $E \not\equiv \mathit{Type}$.

*(iii)* *If* $\Gamma \vdash A = B : E$, *then* $[\![\Gamma]\!] \downarrow$, *and for all* $\rho \in [\![\Gamma]\!]$, $[\![A]\!]_\rho = [\![B]\!]_\rho$, $[\![A]\!]_\rho \in [\![E]\!]_\rho$ *and also* $[\![E]\!]_\rho \in [\![\mathit{Type}]\!]$ *if* $E \not\equiv \mathit{Type}$.

*(iv)* $\nvdash a : \mathbf{0}$. □

## 6 Conclusions and Future Work

We have introduced and formalised a new principle, namely induction-induction, for defining sets in Martin-Löf type theory. The principle allows us to simultaneously introduce $A : \mathrm{Set}$ and $B : A \to \mathrm{Set}$, both defined inductively. This principle is used in recent formulations of the meta-theory of type theory in type theory [5,4].

In the future, the relationship between the principle presented here and what is implemented in Agda will be investigated further. Agda implements arbitrary number of levels, i.e. we can have $A$ : Set, $B : A \to$ Set, $C : (a : A) \to B(a) \to$ Set etc., and induction-induction can be used in conjunction with induction-recursion (with the side effect of a self-referring universe). Apart from this, we speculate that our theory covers what can be defined in Agda. However, just as for ordinary induction, we do not expect dependent pattern matching to follow from our elimination rules without the addition of Streicher's Axiom K [16].

On the theoretical side, work is underway to show that inductive-inductive definitions can be reduced to indexed inductive definitions. This would show that the proof theoretical strength does not increase compared to ordinary induction. Normalisation, decidability of type checking and a categorical semantics similar to initial algebra semantics for ordinary inductive types are other topics left for future work.

**Acknowledgements.** We wish to thank Phillip James and the anonoymous referees of this and previous versions of this article for their helpful comments.

## References

1. Aczel, P.: On relating type theories and set theories. In: Altenkirch, T., Naraschewski, W., Reus, B. (eds.) TYPES 1998. LNCS, vol. 1657, pp. 1–18. Springer, Heidelberg (1999)
2. Backhouse, R., Chisholm, P., Malcolm, G., Saaman, E.: Do-it-yourself type theory. Formal Aspects of Computing 1(1), 19–84 (1989)
3. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nordic Journal of Computing 10, 265–269 (2003)
4. Chapman, J.: Type theory should eat itself. Electronic Notes in Theoretical Computer Science 228, 21–36 (2009)
5. Danielsson, N.: A formalisation of a dependently typed language as an inductive-recursive family. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 93–109. Springer, Heidelberg (2007)
6. Dybjer, P.: Inductive families. Formal aspects of computing 6(4), 440–465 (1994)
7. Dybjer, P.: Internal type theory. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 120–134. Springer, Heidelberg (1996)
8. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. Journal of Symbolic Logic 65(2), 525–549 (2000)
9. Dybjer, P., Setzer, A.: A finite axiomatization of inductive-recursive definitions. In: Girard, J. (ed.) TLCA 1999. LNCS, vol. 1581, pp. 129–146. Springer, Heidelberg (1999)
10. Dybjer, P., Setzer, A.: Induction–recursion and initial algebras. Annals of Pure and Applied Logic 124(1-3), 1–47 (2003)
11. Dybjer, P., Setzer, A.: Indexed induction–recursion. Journal of logic and algebraic programming 66(1), 1–49 (2006)
12. Martin-Löf, P.: Intuitionistic type theory. Bibliopolis Naples (1984)
13. Morris, P.: Constructing Universes for Generic Programming. Ph.D. thesis, University of Nottingham (2007)
14. Nordvall Forsberg, F., Setzer, A.: Induction-induction: Agda development and extended version (2010), `http://cs.swan.ac.uk/~csfnf/induction-induction/`
15. Palmgren, E.: On universes in type theory. In: Sambin, G., Smith, J. (eds.) Twenty five years of constructive type theory, pp. 191–204. Oxford University Press, Oxford (1998)
16. Streicher, T.: Investigations into intensional type theory. Habilitiation Thesis (1993)
17. The Agda Team: The Agda wiki (2010), `http://wiki.portal.chalmers.se/agda/`