

A polymorphic representation of induction-recursion

Venanzio Capretta

March 29, 2004

Dybjer's simultaneous inductive-recursive definitions [3] can be represented in the Calculus of Inductive Constructions by an impredicative Σ -type. The idea was suggested by the presentation of inductive types à la Mendler given by Uustalu and Vene [6], in particular by their example implementation of Fibonacci sequence.

In general, given an inductive-recursive definition:

$$\begin{aligned} T &: (\Gamma) s \\ c_i &: (\Theta_i[T, g])(T \bar{t}_i) \\ g &: (\Gamma)(T \bar{x}) \rightarrow A \\ g(c_i \bar{z}_i) &= e[g, \bar{z}_i] \end{aligned}$$

where s is a sort, \bar{x} is the sequence of variables assumed in Γ , \bar{z} is the sequence of variables assumed in Θ_i , we translate it by replacing the occurrences of T and g in the types of the constructors by variables that are abstracted in each of the constructors:

$$\begin{aligned} T^* &: (\Gamma) s \\ c_i^* &: (Y: (\Gamma) s; f: (\Gamma)(Y \bar{x}) \rightarrow A; \Theta_i[Y, f])(T^* \bar{t}_i) \\ g^* &: (\Gamma)(T^* \bar{x}) \rightarrow A \\ g^*(c_i Y f \bar{z}_i) &= e[f, \bar{z}_i] \end{aligned}$$

The family T^* does not depend on the function g^* , because all calls to it have been replaced by f , and is not inductive, because all recursive occurrences of T in the constructors have been replaced by Y . Therefore, the definition is a standard polymorphic definition in the Calculus of Inductive Constructions and g^* is a standard function on it defined by case analysis. However, we have enlarged the set of elements of T by polymorphically generalizing T to a variable Y . We can restrict it to the original desired elements by an inductive predicate on T^* .

We illustrate the method with a simple example. More examples follow the exposition of the general method.

1 A inductive-recursive type of trees

Consider the following inductive-recursive definition:

$$\begin{aligned}
\mathbb{T}: \text{Set} \\
\text{lf}: \mathbb{T} \\
\text{nd}: (t: \mathbb{T}) \mathbb{T}^{(\text{nds } t)} \rightarrow \mathbb{T} \\
\text{nds}: \mathbb{T} \rightarrow \mathbb{N} \\
\text{nds lf} = 0 \\
\text{nds } (\text{nd } t \bar{v}) = \sum_{z \in \bar{v}} (\text{nds } z) + 1
\end{aligned}$$

Intuitively, T is a type of trees with a restriction on the number of children of a parent node. The constructor lf gives a leaf. The constructor nd gives a node with at least one child. If t is the first child, then the number of extra children must be equal to the result of the function nds on t . The function nds simply counts the nodes of a tree that are not on the first child of a node.

The idea of the representation consists in substituting the references to \mathbb{T} and nds in the types of the constructors by polymorphically quantified variables. Intuitively, we think of the type variable Y as representing a subset of \mathbb{T} , $Y \subseteq \mathbb{T}$ and we think of the variable $f: Y \rightarrow \mathbb{N}$ as being the restriction of nds to Y . The new definition then corresponds to restricting the recursive arguments of constructors to a subset of \mathbb{T} . The definition becomes then:

$$\begin{aligned}
\mathbb{T}^*: \text{Set} \\
\text{lf}^*: \mathbb{T}^* \\
\text{nd}^*: (Y: \text{Set}; f: Y \rightarrow \mathbb{N}; t: Y) Y^{(f \ t)} \rightarrow \mathbb{T}^* \\
\text{nds}^*: \mathbb{T}^* \rightarrow \mathbb{N} \\
\text{nds lf}^* = 0 \\
\text{nds } (\text{nd}^* Y f t \bar{v}) = \sum_{z \in \bar{v}} (f \ z) + 1
\end{aligned}$$

Since the call of the function nds in the type of the constructor nd has been replaced by a call to the variable f , the definition is no longer inductive-recursive. Moreover, recursive arguments of the constructor nds have been replaced by arguments in Y , so the definition of T^* is not inductive, but it is simply a disjunction type. Notice that the function nds^* is not even recursive, since in its body it doesn't call itself but just the variable f .

However, the type \mathbb{T}^* is too big with respect to the original desired type of trees: Since we can use any type Y for the constructors, a parent node may have children belonging to any type. We want to restrict the type so that only other trees of the same kind are allowed as children. We do this by an inductive predicate on \mathbb{T}^* that mimic the original definition of \mathbb{T} :

$$\begin{aligned}
\mathbb{T}_{\text{check}}: \mathbb{T}^* \rightarrow \text{Prop} \\
\text{lf}_{\text{check}}: (\mathbb{T}_{\text{check}} \text{ lf}^*) \\
\text{nd}_{\text{check}}: (t: \mathbb{T}^*; h_t: (\mathbb{T}_{\text{check}} \ t); \bar{v}: \mathbb{T}^{*(\text{nds } t)}; \overline{h_v}: (\mathbb{T}_{\text{check}} \ \bar{v})) (\mathbb{T}_{\text{check}} \ (\text{nd}^* \ \mathbb{T}^* \ \text{nds}^* \ t \ \bar{v}))
\end{aligned}$$

where we denoted by $\overline{h_v}: (\mathbb{T}_{\text{check}} \ \bar{v})$ the assumption of a variable $h_{v_k}: (\mathbb{T}_{\text{check}} \ v_k)$ for every element v_k of the vector \bar{v} .

Now we can recover the originally desired type and function by:

$$\begin{aligned} \mathbf{T} &= \Sigma_{x:\mathbf{T}^*}(\mathbf{T}_{\text{check}} x) \\ \mathbf{nds} t &= (\mathbf{nds}^* (\pi_1 t)) \end{aligned}$$

We obtain the original constructors as functions on \mathbf{T} :

$$\begin{aligned} \text{lf} &= \langle \text{lf}^*, \text{lf}_{\text{check}} \rangle \\ (\mathbf{nd} t \bar{v}) &= \langle (\mathbf{nd}^* \mathbf{T}^* \mathbf{nds}^* (\pi_1 t) \overline{(\pi_1 v)}), (\mathbf{nd}_{\text{check}} (\pi_1 t) (\pi_2 t) \overline{(\pi_1 v)} \overline{(\pi_2 v)}) \rangle \end{aligned}$$

where $\overline{(\pi_i v)}$ denotes the operation of mapping π_i on every element of the vector \bar{v} .

With these definitions the recursive equations for \mathbf{nds} can be verified:

$$\begin{aligned} \mathbf{nds} \text{lf} &= (\mathbf{nds}^* (\pi_1 \langle \text{lf}^*, \text{lf}_{\text{check}} \rangle)) \\ &= (\mathbf{nds}^* \text{lf}^*) = 0 \\ \mathbf{nds} (\mathbf{nd} t \bar{v}) &= (\mathbf{nds}^* (\pi_1 \langle (\mathbf{nd}^* \mathbf{T}^* \mathbf{nds}^* (\pi_1 t) \overline{(\pi_1 v)}), \\ &\quad (\mathbf{nd}_{\text{check}} (\pi_1 t) (\pi_2 t) \overline{(\pi_1 v)} \overline{(\pi_2 v)}) \rangle)) \\ &= (\mathbf{nds}^* (\mathbf{nd}^* \mathbf{T}^* \mathbf{nds}^* (\pi_1 t) \overline{(\pi_1 v)})) = \sum_{z \in \bar{v}} (\mathbf{nds}^* (\pi_1 z)) + 1 \\ &= \sum_{z \in \bar{v}} (\mathbf{nds} z) + 1 \end{aligned}$$

2 Inductive-recursive definitions

We recall the formal definition of simultaneous induction-recursion [3] and define the translation into the Calculus of Inductive Constructions.

Formation rules The general formation rules for a simultaneous inductive-recursive definition are:

$$\begin{aligned} \mathbf{P}: (\mathbf{a}::\Gamma)\mathbf{Set} \\ \mathbf{f}: (\mathbf{a}::\Gamma)(c:(\mathbf{P} \mathbf{a}))\psi[\mathbf{a}] \end{aligned}$$

where $\Gamma \equiv a_1:A_1, \dots, a_n:A_n$ is a valid context and $\Gamma \vdash \psi[\mathbf{a}]:\mathbf{Set}$. The general definition by Dybjer has $\psi[\mathbf{a}]:\mathbf{Type}$, but the representation in the Calculus of Inductive Constructions works only if the result type of the function is small. One consequence of this restriction is that we will not be able to use the translation to define universes.

Next we define the notion of a valid recursive context with respect to \mathbf{P} and \mathbf{f} . This is a context $\Theta \equiv t_1:\theta_1, \dots, t_m:\theta_m$ depending on the assumptions of \mathbf{P} and \mathbf{f} that can be used to define the arguments for a constructor of \mathbf{P} :

$$c: (\mathbf{t}::\Theta)(\mathbf{P} \mathbf{q}).$$

We will also give conditions under which the sequence of terms \mathbf{q} is valid. The restrictions that we impose on Θ are meant to restrict the occurrences of \mathbf{P} and \mathbf{f} so that the terms constructed by \mathbf{c} are well-founded.

We distinguish in Θ two kinds of assumptions, recursive and non-recursive.

- A non-recursive premise has the form $b:\beta$ in which P does not occur, but f may occur. We will impose restrictions on how f may occur.
- A recursive premise has the form $u:(\mathbf{x}::\Xi)(P \mathbf{p}[\mathbf{x}])$ where the explicitly shown occurrence of P is the only one. We will impose the same restrictions on Ξ that we imposed on β for non-recursive premises, and give restrictions for \mathbf{p} .

Finally, the arguments \mathbf{q} of the result of the constructor will have the same restrictions as \mathbf{p} for the recursive premises.

Let us first give an intuitive explanation of why we impose these restrictions. They are intended to specify that f can occur only applied to terms that are structurally simpler than the term constructed by c , that is, f can only be applied to the recursive arguments in Θ . We formalize this by saying that, if there was a previous recursive premise $u':(\mathbf{x}'::\Xi')(P \mathbf{p}'[\mathbf{x}'])$, then f can be used in a subexpression of the form $(f \mathbf{p}'[\mathbf{e}]) (u' \mathbf{e})$ for any sequence \mathbf{e} of the right type. We express this by giving the name v' to all these recursive calls: $v':(\mathbf{x}'::\Xi')\psi[\mathbf{p}'[\mathbf{x}']]$. We then state that an assumption or a term is valid if it is obtained from an assumption or a term depending on v' instead of u' by the substitution $v := [\mathbf{x}'::\Xi'](f \mathbf{p}'[\mathbf{x}']) (u' \mathbf{x}')$.

Definitions 1 We define when a context Θ depending on the assumptions P and f is a valid recursive context with respect to P and f . At the same time we define a context $\hat{\Theta}$ that does not depend on P and f ; we identify the sequence \mathbf{u} of recursive premises in Θ and the corresponding sequence \mathbf{v} of premises in $\hat{\Theta}$. We proceed by induction on the length of Θ . We assume that Θ is already a valid context and that we have defined $\hat{\Theta}$, $\mathbf{u} = u_1, \dots, u_k$, and $\mathbf{v} = v_1, \dots, v_k$. Let us call $\rho(\mathbf{u}, \mathbf{v})$ the substitution

$$\rho(\mathbf{u}, \mathbf{v}) = \{v_i := [\mathbf{x}_i::\Xi_i](f \mathbf{p}_i[\mathbf{x}_i]) (u_i \mathbf{x}_i)\}_{i=1, \dots, k}.$$

There are two ways of extending Θ :

- A non-recursive premise has the form:

$$b:\beta \equiv \hat{\beta}[\rho]$$

where $\hat{\beta}$ is any type such that $\hat{\Theta} \vdash \hat{\beta}:\text{Set}$.

In this case we put $\widehat{\Theta}; b:\beta \equiv \hat{\Theta}; b:\hat{\beta}$ and we leave \mathbf{u} and \mathbf{v} unchanged.

- A recursive premise has the form:

$$u:(\mathbf{x}::\Xi)(P \mathbf{p}[\mathbf{x}])$$

where $\Xi \equiv \hat{\Xi}[\rho]$ for any context Ξ valid under the assumptions $\hat{\Theta}$, and $\mathbf{p} \equiv \hat{\mathbf{p}}[\rho]$ for any sequence of terms $\hat{\mathbf{p}}$ such that $\hat{\Theta}; \hat{\Xi} \vdash \mathbf{p}::\Gamma$.

In this case we put $\Theta; u:(\widehat{\Xi})(P \mathbf{p}[\mathbf{x}]) \equiv \hat{\Theta}; v:(\hat{\Xi})\psi[\hat{\mathbf{p}}[\mathbf{x}]]$ and we add u to the list of recursive assumptions and v to their non-recursive counterparts.

We are now ready to give the introduction rules for the inductive-recursive definition.

Constructors: A constructor of the family P has the general form

$$c: (\Theta)(P \mathbf{q})$$

where Θ is a valid recursive context and $\mathbf{q} \equiv \widehat{\mathbf{q}}[\rho]$ for any sequence $\widehat{\mathbf{q}}$ of terms such that $\widehat{\Theta} \vdash \widehat{\mathbf{q}}::\Gamma$.

Finally, we give the general form of the recursive equations for f . We have one equation for every constructor of P and we use the same characterization as for β , Ξ , \mathbf{p} , and \mathbf{q} to specify that the recursive calls are applied to structurally smaller terms.

Recursive equation: There is one recursive equation for every constructor of P of the form

$$(f \mathbf{q} (c \mathbf{t})) = e[\rho]$$

for any term e such that $\widehat{\Theta} \vdash e: \psi[\widehat{\mathbf{q}}]$.

3 Definition of the representation

Given an inductive-recursive definition satisfying the conditions of the previous sections, we show how to represent it in the Calculus of Inductive Constructions. The main idea consists in quantifying the references to the family P and the function f inside the types of the constructors. In what follows we need to consider several of the expressions of the previous section with P and f substituted with variables P and f . We signal this by writing P and f as subscripts. For example we write $\Theta_{P,f}$ for the sequence of assumptions of a constructor with P replaced by P and f replaced by f . We indicate substitution of these variables by changing the subscripts. So Θ_{P^*,f^*} will denote $\Theta_{P,f}[P := P^*; f := f^*]$ and the original Θ will be the same as $\Theta_{P,f}$.

In what follows we then assume that all expressions have been changed by replacing every occurrence of P by a variable P and every occurrence of f by a variable f . First of all we define the corresponding type by quantifying over P and f in the constructor types:

$$\begin{aligned} P^*: P: (\mathbf{a}::\Gamma)\text{Set} \\ c^*: (P: (\mathbf{a}::\Gamma)\text{Set}; f: (\mathbf{a}::\Gamma)(c: (P \mathbf{a}))\psi[\mathbf{a}]; \mathbf{t}::\Theta_{P,f})(P^* \mathbf{q}_{P,f}) \end{aligned}$$

As we said, all occurrences of P and f in Θ and \mathbf{q} have been replaced by P and f . This means that the definition does not depend on f anymore and, since the only occurrence of P^* is in the result type, P^* is not an inductive type, but simply a disjunction type or a Σ -type with named constructors. It is essentially polymorphic because of the second order quantification over P . This impredicativity cannot be avoided by recourse of type universes [4, 5] because we will need to instantiate P to P^* itself.

The function corresponding to f can be defined by case analysis on P^* . The recursive calls are replaced by calls to the variable function f :

$$\begin{aligned} f^* &: (\mathbf{a}::\Gamma)(c: (P^* \mathbf{a}))\psi[\mathbf{a}] \\ (f \mathbf{q} (c^* P f \mathbf{t})) &= e[\rho_{P,f}] \end{aligned}$$

Once again, remember that $\rho_{P,f}$ is obtained by replacing all references to P and f in ρ by P and f .

The next step consists in restricting the allowed elements of P^* to those obtained by applying the constructors only with $P = P^*$ and $f = f^*$. This is done by an inductive predicate P_{check} on P^* that mimic the definition of P . We need to require that all recursive calls must satisfy P_{check} recursively. To this end, we denote by Θ^{check} the sequence of such assumptions for every recursive premise u in Θ . That is, for every recursive assumption $u: (\mathbf{x}::\Xi)(P \mathbf{p}[\mathbf{x}])$ in Θ , corresponding to an assumption $u: (\mathbf{x}::\Xi_{P,f})(P^* \mathbf{p}_{P,f}[\mathbf{x}])$ in $\Theta_{P,f}$, Θ^{check} contains the assumption

$$h_u: (\mathbf{x}::\Xi_{P,f})(P_{\text{check}} (u \mathbf{x})).$$

Then the predicate P_{check} is inductively defined as

$$\begin{aligned} P_{\text{check}} &: (\mathbf{a}::\Gamma)(P^* \mathbf{a}) \rightarrow \text{Prop} \\ C_{\text{check}} &: (\mathbf{t}::\Theta_{P^*f^*}; \Theta^{\text{check}})(P_{\text{check}} \mathbf{q}_{P^*,f^*} (c^* P^* f^* \mathbf{t})) \end{aligned}$$

We can represent P by a family whose elements are pairs of elements of P^* and proofs of P_{check} :

$$\begin{aligned} P &: (\mathbf{a}::\Gamma)\text{Set} \\ P &:= [\mathbf{a}::\Gamma]\Sigma w: (P^* \mathbf{a}).(P_{\text{check}} w) \end{aligned}$$

4 Lists of distinct elements

The following example by Catarina Coquand (see [3] and [1]) is a definition of the type of lists with distinct elements where we define by simultaneous induction-recursion the type and the relation Fresh expressing the fact that an object does not occur as element of a list.

Let $A: \text{Set}$ be the type of elements of the lists and $\sharp: A \rightarrow A \rightarrow \text{Prop}$ be the relation stating when two objects of A are distinct. Then the type of lists of distinct elements is defined as:

$$\begin{aligned} \text{dList} &: \text{Set} \\ \text{dnil} &: \text{dList} \\ \text{dcons} &: (a: A; l: \text{dList}; h: (\text{Fresh } l \ a))\text{dList} \\ \text{Fresh} &: \text{dList} \rightarrow A \rightarrow \text{Prop} \\ \text{Fresh } \text{dnil} &= [x]\top \\ \text{Fresh } (\text{dcons } a \ l \ h) &= [x](x \sharp a) \wedge (\text{Fresh } l \ x) \end{aligned}$$

We proceed as in the previous example by generalizing the references to \mathbf{dList} and \mathbf{Fresh} in the type of the constructors.

$$\begin{aligned} \mathbf{dList}^* &: \mathbf{Set} \\ \mathbf{dnil}^* &: \mathbf{dList} \\ \mathbf{dcons}^* &: (Y : \mathbf{Set}; j : Y \rightarrow \mathbf{dList}^*; f : Y \rightarrow \mathbf{Prop}; a : A; l : Y; h : (f \ l \ y)) \mathbf{dList}^* \end{aligned}$$

The translation of \mathbf{Fresh} is different, because we cannot translate it directly by recursion on \mathbf{dList}^* , since \mathbf{dList}^* has large arguments in the constructors, it is not allowed to eliminate it over large types, and the type of \mathbf{Fresh} , $\mathbf{dList} \rightarrow A \rightarrow \mathbf{Prop}$ is large. Instead we translate \mathbf{Fresh} as an inductive predicate:

$$\begin{aligned} \mathbf{Fresh}^* &: \mathbf{dList} \rightarrow A \rightarrow \mathbf{Prop} \\ \mathbf{fresh}_{\mathbf{nil}} &: (x : A) (\mathbf{Fresh}^* \ \mathbf{dnil}^* \ x) \\ \mathbf{fresh}_{\mathbf{cons}} &: (Y : \mathbf{Set}; j : Y \rightarrow \mathbf{dList}; Fr : Y \rightarrow A \rightarrow \mathbf{Set}; \\ & \quad y : A; l : Y; h : (Fr \ l \ y); x : A) \\ & \quad (x \# y) \rightarrow (Fr \ l \ x) \rightarrow (\mathbf{Fresh}^* \ (\mathbf{dcons}^* \ Y \ j \ Fr \ y \ l \ h) \ x) \end{aligned}$$

Once again, the type \mathbf{dList}^* is too large and we need to restrict it by an inductive predicate.

$$\begin{aligned} \mathbf{Fresh}_{\mathbf{check}} &: \mathbf{dList} \rightarrow \mathbf{Prop} \\ \mathbf{nil}_{\mathbf{check}} &: (\mathbf{Fresh}_{\mathbf{check}} \ \mathbf{dnil}) \\ \mathbf{cons}_{\mathbf{check}} &: (y : A; l : \mathbf{dList}; h : (\mathbf{Fresh}_{\mathbf{check}} \ l); p : (\mathbf{Fresh}^* \ l \ y)) \\ & \quad (\mathbf{Fresh}_{\mathbf{check}} \ (\mathbf{dcons} \ \mathbf{dList} \ \lambda z.z \ \mathbf{Fresh}^* \ y \ l \ p)) \end{aligned}$$

Then the original type, function, and constructors can be defined as:

$$\begin{aligned} \mathbf{dList} &= \Sigma_{l : \mathbf{dList}^*} (\mathbf{Fresh}_{\mathbf{check}} \ l) \\ \mathbf{Fresh} \ l \ a &= (\mathbf{Fresh}^* \ (\pi_1 \ l) \ a) \\ \mathbf{dnil} &= \langle \mathbf{dnil}^*, \mathbf{nil}_{\mathbf{check}} \rangle \\ \mathbf{dcons} \ a \ l \ p &= \langle (\mathbf{dcons}^* \ \mathbf{dList}^* \ \lambda z.z \ \mathbf{Fresh}^* \ a \ (\pi_1 \ l) \ p), (\mathbf{cons}_{\mathbf{check}} \ a \ (\pi_1 \ l) \ (\pi_2 \ l) \ p) \rangle \end{aligned}$$

5 Representation of nested recursive functions

The same translation can be applied to the Bove-Capretta method to untangle the definition of function from the accessibility predicate. We illustrate it on the most simple example, the \mathbf{nest} function [2], which is defined using induction-recursion as:

$$\begin{aligned} \mathbf{D}_{\mathbf{nest}} &: \mathbb{N} \rightarrow \mathbf{Prop} \\ \mathbf{nest}_0 &: (\mathbf{D}_{\mathbf{nest}} \ 0) \\ \mathbf{nest}_S &: (n : \mathbb{N}; h_1 : (\mathbf{D}_{\mathbf{nest}} \ n); h_2 : (\mathbf{D}_{\mathbf{nest}} \ (\mathbf{nest} \ n \ h_1))) (\mathbf{D}_{\mathbf{nest}} \ (S \ n)) \\ \mathbf{nest} &: (n : \mathbb{N}) (\mathbf{D}_{\mathbf{nest}} \ n) \rightarrow \mathbb{N} \\ \mathbf{nest} \ 0 &= 0 \\ \mathbf{nest} \ (S \ n) \ (\mathbf{nest}_S \ n \ h_1 \ h_2) &= (\mathbf{nest} \ (\mathbf{nest} \ n \ h_1) \ h_2) \end{aligned}$$

Proceeding as in the two previous examples, we get the impredicative representation:

$$\begin{aligned}
D_{\text{nest}}^* &: \mathbb{N} \rightarrow \mathbf{Prop} \\
\text{nest}_0^* &: (D_{\text{nest}}^* 0) \\
\text{nest}_S^* &: (Y: \mathbb{N} \rightarrow \mathbf{Prop}; j: (x: \mathbb{N})(Y x) \rightarrow (D_{\text{nest}}^* x); f: (x: \mathbb{N})(Y x) \rightarrow \mathbb{N}; \\
&\quad n: \mathbb{N}; h_1: (Y n); h_2: (Y (f n h_1)))(D_{\text{nest}}^* (S n)) \\
\text{nest}^* &: (n: \mathbb{N})(D_{\text{nest}}^* n) \rightarrow \mathbb{N} \\
\text{nest}^* 0 & \text{ nest}_0^* = 0 \\
\text{nest}^* (S n) & (\text{nest}_S^* Y j f n h_1 h_2) = (f (f n h_1) h_2)
\end{aligned}$$

Now we can prove $(D_{\text{nest}}^* n)$ using different predicates Y and functions f , so we need to restrict the possible proofs by a predicate on the domain predicate:

$$\begin{aligned}
\text{Nest}_{\text{check}} &: (n: \mathbb{N})(D_{\text{nest}} n) \rightarrow \mathbf{Prop} \\
0_{\text{check}} &: (\text{Nest}_{\text{check}} 0 \text{ nest}_0^*) \\
S_{\text{check}} &: (n: \mathbb{N}; h_1: (D_{\text{nest}}^* n); k_1: (\text{Nest}_{\text{check}} n h_1); \\
&\quad h_2: (D_{\text{nest}}^* (\text{nest}^* n h_1)); k_2: (\text{Nest}_{\text{check}} (\text{nest}^* n h_1) h_2)) \\
&\quad (\text{Nest}_{\text{check}} (S n) (\text{nest}_S^* D_{\text{nest}}^* \lambda x. \lambda q. q \text{ nest}^* n h_1 h_2))
\end{aligned}$$

The original domain predicate, function, and constructors for the predicate can be defined as:

$$\begin{aligned}
D_{\text{nest}} n &= \Sigma_{z: (D_{\text{nest}}^* n)} (\text{Nest}_{\text{check}} n z) \\
\text{nest} n h &= (\text{nest}^* n (\pi_1 h)) \\
\text{nest}_0 &= \langle \text{nest}_0^*, 0_{\text{check}} \rangle \\
\text{nest}_S n h_1 h_2 &= \langle (\text{nest}_S^* D_{\text{nest}}^* \lambda x. \lambda q. q \text{ nest}^* n (\pi_1 h_1) (\pi_1 h_2)), \\
&\quad (S_{\text{check}} n (\pi_1 h_1) (\pi_2 h_1) (\pi_1 h_2) (\pi_2 h_2)) \rangle
\end{aligned}$$

and the validity of recursion equations for nest can be verified.

References

- [1] F. Blanqui. Inductive types in the Calculus of Algebraic Constructions (extended abstract). In *TLCA '03*, 2003. To appear in LNCS.
- [2] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.
- [3] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.
- [4] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic colloquium '73*, pages 153–175. North-Holland, 1975.

- [5] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.
- [6] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.