

A possible reformulation of polymorphism: the simply typed lambda calculus \mathcal{N}

Structure and Deductions 2017 Workshop
Oxford, September 9, 2017

S. Berardi (*sp.*), U. de' Liguoro

Torino University, Italy



A Possible Reformulation of Polymorphism

- 1 This talk is about representing the constructive content of **second order deduction** (as expressed in system \mathcal{F}) through **uniform operations on tree structures**.
- 2 We investigate the possibility of representing polymorphic maps on data types definable in system \mathcal{F} in a simply typed λ -calculus, system \mathcal{N} .
- 3 Reduction of \mathcal{N} are:
 - **algebraic** reductions
 - reductions for **primitive recursion** on trees
 - reductions for **uniform recursion**

Uniform recursion represents polymorphism through **uniform application**, a concrete operation on trees.

A Possible Reformulation of Polymorphism

- 1 This talk is about representing the constructive content of **second order deduction** (as expressed in system \mathcal{F}) through **uniform operations on tree structures**.
- 2 We investigate the possibility of representing polymorphic maps on data types definable in system \mathcal{F} in a simply typed λ -calculus, system \mathcal{N} .
- 3 Reduction of \mathcal{N} are:
 - **algebraic** reductions
 - reductions for **primitive recursion** on trees
 - reductions for **uniform recursion**

Uniform recursion represents polymorphism through **uniform application**, a concrete operation on trees.

A Possible Reformulation of Polymorphism

- 1 This talk is about representing the constructive content of **second order deduction** (as expressed in system \mathcal{F}) through **uniform operations on tree structures**.
- 2 We investigate the possibility of representing polymorphic maps on data types definable in system \mathcal{F} in a simply typed λ -calculus, system \mathcal{N} .
- 3 Reduction of \mathcal{N} are:
 - **algebraic** reductions
 - reductions for **primitive recursion** on trees
 - reductions for **uniform recursion**

Uniform recursion represents polymorphism through **uniform application**, a concrete operation on trees.

This is an ongoing work!

The result we already have about system \mathcal{N} are:

- 1 **normalization** (with an intuitionistic proof)
- 2 all trees denoted by some term t in some data type D of system \mathcal{N} are **well-founded**.

The result we are checking are:

- 1 system \mathcal{N} defines a **realization model** of second order arithmetic
- 2 **equivalence between system \mathcal{N} and polymorphism**: we may compile \mathcal{F} in \mathcal{N} , and for any data types D, E , system \mathcal{N} and system \mathcal{F} define the same maps $f : D \rightarrow E$

This is an ongoing work!

The result we already have about system \mathcal{N} are:

- 1 **normalization** (with an intuitionistic proof)
- 2 all trees denoted by some term t in some data type D of system \mathcal{N} are **well-founded**.

The result we are checking are:

- 1 system \mathcal{N} defines a **realization model** of second order arithmetic
- 2 **equivalence between system \mathcal{N} and polymorphism**: we may compile \mathcal{F} in \mathcal{N} , and for any data types D, E , system \mathcal{N} and system \mathcal{F} define the same maps $f : D \rightarrow E$

Plan of the Talk

- §1. Data types and Types of system \mathcal{N}
- §2. Primitive Recursion on Trees
- §3. Recursion in system \mathcal{N} : Uniform Recursion
- §4. Definitions and Results for System \mathcal{N}
- §5. Conclusions

§1. Data types and Types of system \mathcal{N}

We define nested data types as in Martin Lof ([1]).

- 1 The set **Data** of data types for trees in \mathcal{N} is inductively defined by $D ::= (D_0, \dots, D_{n-1})$.
- 2 D denotes the set of well-founded trees whose node all have index set for children the set D_i for some $i < n$.
- 3 The set **Tp** of types of \mathcal{N} is inductively defined by $T ::= D | T \times T | T \rightarrow T$ for any data type $D \in \mathbf{Data}$.

§1. Data types and Types of system \mathcal{N}

We define nested data types as in Martin Lof ([1]).

- 1 The set **Data** of data types for trees in \mathcal{N} is inductively defined by $D ::= (D_0, \dots, D_{n-1})$.
- 2 D denotes the set of well-founded trees whose node all have index set for children the set D_i for some $i < n$.
- 3 The set **Tp** of types of \mathcal{N} is inductively defined by $T ::= D | T \times T | T \rightarrow T$ for any data type $D \in \mathbf{Data}$.

§1. Data types and Types of system \mathcal{N}

We define nested data types as in Martin Lof ([1]).

- 1 The set **Data** of data types for trees in \mathcal{N} is inductively defined by $D ::= (D_0, \dots, D_{n-1})$.
- 2 D denotes the set of well-founded trees whose node all have index set for children the set D_i for some $i < n$.
- 3 The set **Tp** of types of \mathcal{N} is inductively defined by $T ::= D | T \times T | T \rightarrow T$ for any data type $D \in \mathbf{Data}$.

Tree constructors

- 1 Tree constructors for $D = (D_1, \dots, D_n)$ are constants $\mathbf{c}_i : (D_i \rightarrow D) \rightarrow D$, one for each $i = 1, \dots, n$.
- 2 We call D_i the index set of \mathbf{c}_i .
- 3 \mathbf{c}_i takes in argument a family of trees in D indexed on D_i , and represented by a term $f : D_i \rightarrow D$.
- 4 \mathbf{c}_i returns the tree $\mathbf{c}_i(f) : D$, whose root has:
 - label \mathbf{c}_i
 - children indexed on D_i (for each $e : D_i$, one child $f(e) : D$).
- 5 The index set D_i of a node (and therefore the arity of the node) may be empty, a singleton, a finite set of any cardinality, or may be infinite.

Tree constructors

- 1 Tree constructors for $D = (D_1, \dots, D_n)$ are constants $\mathbf{c}_i : (D_i \rightarrow D) \rightarrow D$, one for each $i = 1, \dots, n$.
- 2 We call D_i the index set of \mathbf{c}_i .
- 3 \mathbf{c}_i takes in argument a family of trees in D indexed on D_i , and represented by a term $f : D_i \rightarrow D$.
- 4 \mathbf{c}_i returns the tree $\mathbf{c}_i(f) : D$, whose root has:
 - label \mathbf{c}_i
 - children indexed on D_i (for each $e : D_i$, one child $f(e) : D$).
- 5 The index set D_i of a node (and therefore the arity of the node) may be empty, a singleton, a finite set of any cardinality, or may be infinite.

Some Examples of Data Types

All finite sets and **Nat** may be represented in **Data** as follows.

- 1 **Empty set.** Assume $D = () \in \mathbf{Data}$. Then D has no constructors and denotes the empty set \emptyset .
- 2 **Finite sets.** Assume $D = (\emptyset, \dots, \emptyset) \in \mathbf{Data}$ has n constructors $\mathbf{c}_0, \dots, \mathbf{c}_{n-1} : (\emptyset \rightarrow D) \rightarrow D$ with index set empty. (n constant constructors).
- 3 Given a dummy map $\mathbf{dummy} : \emptyset \rightarrow D$, if we abbreviate $i \equiv \mathbf{c}_i(\mathbf{dummy})$ for all $i < n$, then $D = \{0, \dots, n-1\}$.
- 4 **Natural Numbers.** Assume $D = (\emptyset, \{0\}) \in \mathbf{Data}$. Then D has one constructor \mathbf{c}_0 with empty index set (a constant constructor) and one constructor \mathbf{c}_1 with singleton index set (a unary constructor).
- 5 If we abbreviate $0 \equiv \mathbf{c}_0(\mathbf{dummy})$ and $x + 1 \equiv \mathbf{c}_1(f)$ with $f(0) = x$, we may prove that D is isomorphic to **Nat**.

Some Examples of Data Types

All finite sets and **Nat** may be represented in **Data** as follows.

- 1 **Empty set.** Assume $D = () \in \mathbf{Data}$. Then D has no constructors and denotes the empty set \emptyset .
- 2 **Finite sets.** Assume $D = (\emptyset, \dots, \emptyset) \in \mathbf{Data}$ has n constructors $\mathbf{c}_0, \dots, \mathbf{c}_{n-1} : (\emptyset \rightarrow D) \rightarrow D$ with index set empty. (n constant constructors).
- 3 Given a dummy map **dummy** : $\emptyset \rightarrow D$, if we abbreviate $i \equiv \mathbf{c}_i(\mathbf{dummy})$ for all $i < n$, then $D = \{0, \dots, n - 1\}$.
- 4 **Natural Numbers.** Assume $D = (\emptyset, \{0\}) \in \mathbf{Data}$. Then D has one constructor \mathbf{c}_0 with empty index set (a constant constructor) and one constructor \mathbf{c}_1 with singleton index set (a unary constructor).
- 5 If we abbreviate $0 \equiv \mathbf{c}_0(\mathbf{dummy})$ and $x + 1 \equiv \mathbf{c}_1(f)$ with $f(0) = x$, we may prove that D is isomorphic to **Nat**.

Some Examples of Data Types

All finite sets and **Nat** may be represented in **Data** as follows.

- 1 **Empty set.** Assume $D = () \in \mathbf{Data}$. Then D has no constructors and denotes the empty set \emptyset .
- 2 **Finite sets.** Assume $D = (\emptyset, \dots, \emptyset) \in \mathbf{Data}$ has n constructors $\mathbf{c}_0, \dots, \mathbf{c}_{n-1} : (\emptyset \rightarrow D) \rightarrow D$ with index set empty. (n constant constructors).
- 3 Given a dummy map **dummy** : $\emptyset \rightarrow D$, if we abbreviate $i \equiv \mathbf{c}_i(\mathbf{dummy})$ for all $i < n$, then $D = \{0, \dots, n - 1\}$.
- 4 **Natural Numbers.** Assume $D = (\emptyset, \{0\}) \in \mathbf{Data}$. Then D has one constructor \mathbf{c}_0 with empty index set (a constant constructor) and one constructor \mathbf{c}_1 with singleton index set (a unary constructor).
- 5 If we abbreviate $0 \equiv \mathbf{c}_0(\mathbf{dummy})$ and $x + 1 \equiv \mathbf{c}_1(f)$ with $f(0) = x$, we may prove that D is isomorphic to **Nat**.

More examples of Data Types

- 1 **Binary trees.** Assume $D = (\emptyset, \{0, 1\}) \in \mathbf{Data}$. D has 2 constructors $\mathbf{c}_0, \mathbf{c}_2$ of index sets \emptyset and $\{0, 1\}$ (arity 0 and 2). We claim that D is isomorphic to the set of finite binary trees.
- 2 Indeed, if $f : \emptyset \rightarrow D$ and $g : \{0, 1\} \rightarrow D$ with $g(0) = t_0, g(1) = t_1$, then we may set **leaf** = $\mathbf{c}_0(f)$ and **mktree**(t_1, t_2) = $\mathbf{c}_2(g)$.
- 3 **Well-founded trees.** Assume $D = (\emptyset, \mathbf{Nat}) \in \mathbf{Data}$. We claim that D denotes the set Ω of well-founded trees whose nodes are either leaves or have ω children. D has 2 constructors $\mathbf{c}_0, \mathbf{c}_2$ of index sets \emptyset and \mathbf{Nat} (arity 0 and ω).
- 4 Indeed, if $f : \emptyset \rightarrow D$ and $g : \mathbf{Nat} \rightarrow D$ with $g(n) = t_n$ for all $n \in \mathbf{Nat}$, then we may set **leaf** = $\mathbf{c}_0(f)$ and $\mathbf{c}_2(g)$ = the tree whose n -th child is t_n .

More examples of Data Types

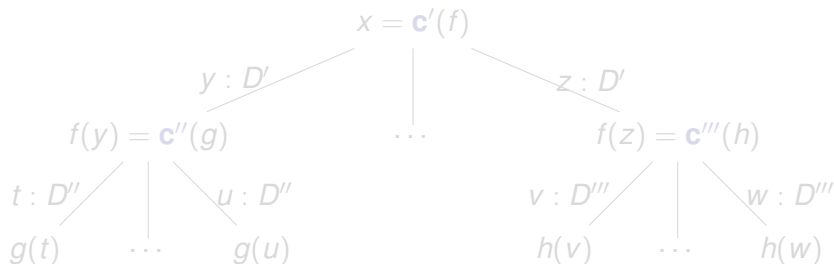
- 1 **Binary trees.** Assume $D = (\emptyset, \{0, 1\}) \in \mathbf{Data}$. D has 2 constructors $\mathbf{c}_0, \mathbf{c}_2$ of index sets \emptyset and $\{0, 1\}$ (arity 0 and 2). We claim that D is isomorphic to the set of finite binary trees.
- 2 Indeed, if $f : \emptyset \rightarrow D$ and $g : \{0, 1\} \rightarrow D$ with $g(0) = t_0, g(1) = t_1$, then we may set $\mathbf{leaf} = \mathbf{c}_0(f)$ and $\mathbf{mktree}(t_1, t_2) = \mathbf{c}_2(g)$.
- 3 **Well-founded trees.** Assume $D = (\emptyset, \mathbf{Nat}) \in \mathbf{Data}$. We claim that D denotes the set Ω of well-founded trees whose nodes are either leaves or have ω children. D has 2 constructors $\mathbf{c}_0, \mathbf{c}_2$ of index sets \emptyset and \mathbf{Nat} (arity 0 and ω).
- 4 Indeed, if $f : \emptyset \rightarrow D$ and $g : \mathbf{Nat} \rightarrow D$ with $g(n) = t_n$ for all $n \in \mathbf{Nat}$, then we may set $\mathbf{leaf} = \mathbf{c}_0(f)$ and $\mathbf{c}_2(g) =$ the tree whose n -th child is t_n .

The general pattern for a Tree

Assume we have a data type $D = (D', D'', D''')$ and

- 1 constructors c', c'', c''' of index sets D', D'', D'''
- 2 index maps f, g, h of type $D' \rightarrow D, D'' \rightarrow D, D''' \rightarrow D$
- 3 indexes $y, z : D', t, u : D'', v, w : D'''$

Assume that $f(y), f(z)$ have values $c''(g), c'''(h)$. Then $x = c'(f) : D$ denotes the tree:

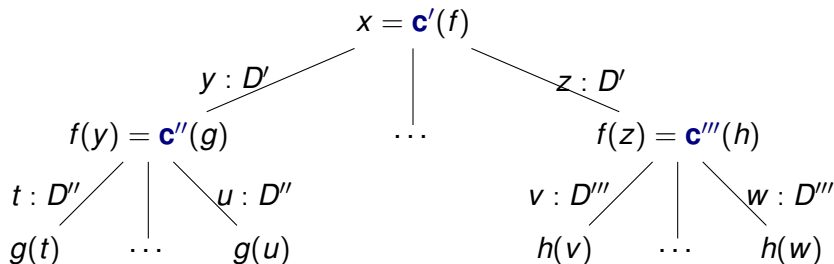


The general pattern for a Tree

Assume we have a data type $D = (D', D'', D''')$ and

- 1 constructors c', c'', c''' of index sets D', D'', D'''
- 2 index maps f, g, h of type $D' \rightarrow D, D'' \rightarrow D, D''' \rightarrow D$
- 3 indexes $y, z : D', t, u : D'', v, w : D'''$

Assume that $f(y), f(y)$ have values $c''(g), c'''(h)$. Then $x = c'(f) : D$ denotes the tree:



- §1. *Data types and Types of system \mathcal{N}*
- **§2. Primitive Recursion on Trees**
- §3. *Recursion in system \mathcal{N} : Uniform Recursion*
- §4. *Definitions and Results for System \mathcal{N}*
- §5. *Conclusions*

§2. Primitive Recursion on Trees

Let $D \in \mathbf{Data}$, $A \in \mathbf{Tp}$. We recall primitive recursive definitions for maps $h : D \rightarrow A$, then we explain how to express them in \mathbf{Data} .

- 1 Assume $D = \emptyset$. Then recursion defines a single map **dummy** : $\emptyset \rightarrow A$.
- 2 Assume $D = \{0, \dots, n-1\}$. Given $r_1, \dots, r_n : A$ we may define $h(i) = r_i$ for all $i < n$.
- 3 Assume $D = \mathbf{Nat}$. Given $r_0 : A$ and $r_1 : A \rightarrow A$ we may define $h(0) = r_0$ and $h(x+1) = r_1(h(x))$ for all $x \in \mathbf{Nat}$.
- 4 Assume $D =$ the set of binary trees, with constructors $\mathbf{c}_0, \mathbf{c}_2(\cdot, \cdot)$. Given $r_0 : A$ and $r_2 : A, A \rightarrow A$ we may define $h(\mathbf{c}_0) = r_0$ and $h(\mathbf{c}_2(t_1, t_2)) = r_2(h(t_1), h(t_2))$.

§2. Primitive Recursion on Trees

Let $D \in \mathbf{Data}$, $A \in \mathbf{Tp}$. We recall primitive recursive definitions for maps $h : D \rightarrow A$, then we explain how to express them in \mathbf{Data} .

- 1 Assume $D = \emptyset$. Then recursion defines a single map **dummy** : $\emptyset \rightarrow A$.
- 2 Assume $D = \{0, \dots, n-1\}$. Given $r_1, \dots, r_n : A$ we may define $h(i) = r_i$ for all $i < n$.
- 3 Assume $D = \mathbf{Nat}$. Given $r_0 : A$ and $r_1 : A \rightarrow A$ we may define $h(0) = r_0$ and $h(x+1) = r_1(h(x))$ for all $x \in \mathbf{Nat}$.
- 4 Assume $D =$ the set of binary trees, with constructors $\mathbf{c}_0, \mathbf{c}_2(\cdot, \cdot)$. Given $r_0 : A$ and $r_2 : A, A \rightarrow A$ we may define $h(\mathbf{c}_0) = r_0$ and $h(\mathbf{c}_2(t_1, t_2)) = r_2(h(t_1), h(t_2))$.

§2. Primitive Recursion on Trees

Let $D \in \mathbf{Data}$, $A \in \mathbf{Tp}$. We recall primitive recursive definitions for maps $h : D \rightarrow A$, then we explain how to express them in \mathbf{Data} .

- 1 Assume $D = \emptyset$. Then recursion defines a single map **dummy** : $\emptyset \rightarrow A$.
- 2 Assume $D = \{0, \dots, n-1\}$. Given $r_1, \dots, r_n : A$ we may define $h(i) = r_i$ for all $i < n$.
- 3 Assume $D = \mathbf{Nat}$. Given $r_0 : A$ and $r_1 : A \rightarrow A$ we may define $h(0) = r_0$ and $h(x+1) = r_1(h(x))$ for all $x \in \mathbf{Nat}$.
- 4 Assume $D =$ the set of binary trees, with constructors $\mathbf{c}_0, \mathbf{c}_2(\cdot, \cdot)$. Given $r_0 : A$ and $r_2 : A, A \rightarrow A$ we may define $h(\mathbf{c}_0) = r_0$ and $h(\mathbf{c}_2(t_1, t_2)) = r_2(h(t_1), h(t_2))$.

§2. Primitive Recursion on Trees

Let $D \in \mathbf{Data}$, $A \in \mathbf{Tp}$. We recall primitive recursive definitions for maps $h : D \rightarrow A$, then we explain how to express them in \mathbf{Data} .

- 1 Assume $D = \emptyset$. Then recursion defines a single map **dummy** : $\emptyset \rightarrow A$.
- 2 Assume $D = \{0, \dots, n-1\}$. Given $r_1, \dots, r_n : A$ we may define $h(i) = r_i$ for all $i < n$.
- 3 Assume $D = \mathbf{Nat}$. Given $r_0 : A$ and $r_1 : A \rightarrow A$ we may define $h(0) = r_0$ and $h(x+1) = r_1(h(x))$ for all $x \in \mathbf{Nat}$.
- 4 Assume $D =$ the set of binary trees, with constructors $\mathbf{c}_0, \mathbf{c}_2(\cdot, \cdot)$. Given $r_0 : A$ and $r_2 : A, A \rightarrow A$ we may define $h(\mathbf{c}_0) = r_0$ and $h(\mathbf{c}_2(t_1, t_2)) = r_2(h(t_1), h(t_2))$.

How to express Primitive Recursion in **Data**

All primitive recursions for some $D \in \mathbf{Data}$ are instances of a single notion of recursion.

- 1 The general pattern for primitive recursion on D is: assume that D has a constructor \mathbf{c}_i with argument list d_1, \dots, d_n, \dots . We first apply h to each d_1, \dots, d_n, \dots , obtaining $h(d_1), \dots, h(d_n), \dots : A$, then we define

$$\mathbf{h}(\mathbf{c}_i(\mathbf{d}_1, \dots, \mathbf{d}_n, \dots)) = \mathbf{r}_i(\mathbf{h}(\mathbf{d}_1), \dots, \mathbf{h}(\mathbf{d}_n), \dots)$$

r_i may **act separately** on each value $h(d_i)$.

- 2 In any $D \in \mathbf{Data}$, the list d_1, \dots, d_n, \dots of arguments of \mathbf{c}_i is replaced by an index map $f : D_j \rightarrow D$ such that $f(e_1) = d_1, \dots, f(e_n) = d_n, \dots$ for some $e_1, \dots, e_n, \dots : D_j$.
- 3 Thus, instead of forming $h(d_1), \dots, h(d_n), \dots : A$, we form $h \circ f : D_j \rightarrow A$, an index map for $h(d_1) = h(f(e_1)), \dots, h(d_n) = h(f(e_n)), \dots : A$. Then we define

$$\mathbf{h}(\mathbf{c}_i(\mathbf{f})) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$$



How to express Primitive Recursion in **Data**

All primitive recursions for some $D \in \mathbf{Data}$ are instances of a single notion of recursion.

- 1 The general pattern for primitive recursion on D is: assume that D has a constructor \mathbf{c}_i with argument list d_1, \dots, d_n, \dots . We first apply h to each d_1, \dots, d_n, \dots , obtaining $h(d_1), \dots, h(d_n), \dots : A$, then we define

$$\mathbf{h}(\mathbf{c}_i(\mathbf{d}_1, \dots, \mathbf{d}_n, \dots)) = \mathbf{r}_i(\mathbf{h}(\mathbf{d}_1), \dots, \mathbf{h}(\mathbf{d}_n), \dots)$$

r_i may **act separately** on each value $h(d_j)$.

- 2 In any $D \in \mathbf{Data}$, the list d_1, \dots, d_n, \dots of arguments of \mathbf{c}_i is replaced by an index map $f : D_j \rightarrow D$ such that $f(e_1) = d_1, \dots, f(e_n) = d_n, \dots$ for some $e_1, \dots, e_n, \dots : D_j$.
- 3 Thus, instead of forming $h(d_1), \dots, h(d_n), \dots : A$, we form $h \circ f : D_j \rightarrow A$, an index map for $h(d_1) = h(f(e_1)), \dots, h(d_n) = h(f(e_n)), \dots : A$. Then we define

$$\mathbf{h}(\mathbf{c}_i(\mathbf{f})) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$$

A General Pattern for Primitive Recursion

Summing up:

- 1 Primitive recursion on $D \in \mathbf{Data}$ defines a map $h : D \rightarrow A$ using one clause r_i for each D_i .
- 2 The clause r_i has type $(D_i \rightarrow A) \rightarrow A$ and it is used on trees $t : D$ of the form $t \equiv \mathbf{c}_i(f)$, for some $f : D_i \rightarrow D$.
- 3 We assume that h is already defined on $f(e)$ for all $e : D_i$, we apply h to the result of f forming $h \circ f : D_i \rightarrow A$, then we apply r_i to the result, obtaining $h(t) = r_i(h \circ f)$.

System \mathcal{N} has a stronger version of recursion which we call **uniform recursion**. Uniform recursion includes all clauses of primitive recursion, plus one extra recursive clause, used when we want to extend h to a larger domain.

A General Pattern for Primitive Recursion

Summing up:

- 1 Primitive recursion on $D \in \mathbf{Data}$ defines a map $h : D \rightarrow A$ using one clause r_i for each D_i .
- 2 The clause r_i has type $(D_i \rightarrow A) \rightarrow A$ and it is used on trees $t : D$ of the form $t \equiv \mathbf{c}_i(f)$, for some $f : D_i \rightarrow D$.
- 3 We assume that h is already defined on $f(e)$ for all $e : D_i$, we apply h to the result of f forming $h \circ f : D_i \rightarrow A$, then we apply r_i to the result, obtaining $h(t) = r_i(h \circ f)$.

System \mathcal{N} has a stronger version of recursion which we call **uniform recursion**. Uniform recursion includes all clauses of primitive recursion, plus one extra recursive clause, used when we want to extend h to a larger domain.

- §1. *Data types and Types of system \mathcal{N}*
- §2. *Primitive Recursion on Trees*
- **§3. Recursion in system \mathcal{N} : Uniform Recursion**
- §4. *Definitions and Results for System \mathcal{N}*
- §5. *Conclusions*

§3. Recursion in system \mathcal{N} : Uniform Recursion

In order to define uniform recursion (which is recursion for system \mathcal{N}), our **first step** is to introduce an operation extending a data type.

- 1 We define an operation $(.)@E$ adding one index set $E \in \mathbf{Data}$ to a data structure $D = (D_0, \dots, D_{n-1})$:
 $D@E = (D_0, \dots, D_{n-1}, E) \in \mathbf{Data}$.
- 2 $D@E$ has one tree constructor $\mathbf{c}_n : (E \rightarrow D@E) \rightarrow D@E$ more than D .
- 3 $(.)@D$ is extended pointwise and componentwise to all types in \mathbf{Tp} by:
 - $(A \times B)@E \equiv A@E \times B@E \in \mathbf{Tp}$
 - $(A \rightarrow B)@E \equiv A \rightarrow B@E \in \mathbf{Tp}$.
- 4 We call the type $A@E$ an *extension* of the type A .

§3. Recursion in system \mathcal{N} : Uniform Recursion

In order to define uniform recursion (which is recursion for system \mathcal{N}), our **first step** is to introduce an operation extending a data type.

- 1 We define an operation $(.)@E$ adding one index set $E \in \mathbf{Data}$ to a data structure $D = (D_0, \dots, D_{n-1})$:
 $D@E = (D_0, \dots, D_{n-1}, E) \in \mathbf{Data}$.
- 2 $D@E$ has one tree constructor $\mathbf{c}_n : (E \rightarrow D@E) \rightarrow D@E$ more than D .
- 3 $(.)@D$ is extended pointwise and componentwise to all types in \mathbf{Tp} by:
 - $(A \times B)@E \equiv A@E \times B@E \in \mathbf{Tp}$
 - $(A \rightarrow B)@E \equiv A \rightarrow B@E \in \mathbf{Tp}$.
- 4 We call the type $A@E$ an *extension* of the type A .

Future constructors

In order to define uniform recursion, our **second step** is to introduce constants **future** $_{m,E,D} : (E \rightarrow D) \rightarrow D$ we call *future constructors*. E is a **term constant** but acts as a type variable.

- 1 The future constructor **future** represents (an approximation of) the new constructor **c_n** : $(E \rightarrow D @ E) \rightarrow D @ E$ we will add in last position to $D = (D_0, \dots, D_{n-1})$ when forming $D @ E = (D_0, \dots, D_{n-1}, E)$.
- 2 We call E a pending extension.
- 3 We add a unary **term operator Forth** $_{m,E}$, executing a pending extension, and acting like a type substitution.
- 4 **Forth** $_{m,E}$ replaces **future** $_{m,E,D}$ with **c_n**:

$$\text{Forth}_{m,E}(\text{future}_{m,E,D}(f)) = \mathbf{c}_n(\text{Forth}_{m,E}(f))$$

- 5 **future** $_{m,E,D}$, **Forth** $_{m,E}$ are extended to all types point-wise and component-wise.

Future constructors

In order to define uniform recursion, our **second step** is to introduce constants **future** $_{m,E,D} : (E \rightarrow D) \rightarrow D$ we call *future constructors*. E is a **term constant** but acts as a type variable.

- 1 The future constructor **future** represents (an approximation of) the new constructor $\mathbf{c}_n : (E \rightarrow D @ E) \rightarrow D @ E$ we will add in last position to $D = (D_0, \dots, D_{n-1})$ when forming $D @ E = (D_0, \dots, D_{n-1}, E)$.
- 2 We call E a pending extension.
- 3 We add a unary **term operator** **Forth** $_{m,E}$, executing a pending extension, and acting like a type substitution.
- 4 **Forth** $_{m,E}$ replaces **future** $_{m,E,D}$ with \mathbf{c}_n :

$$\mathbf{Forth}_{m,E}(\mathbf{future}_{m,E,D}(f)) = \mathbf{c}_n(\mathbf{Forth}_{m,E}(f))$$

- 5 **future** $_{m,E,D}$, **Forth** $_{m,E}$ are extended to all types point-wise and component-wise.

Future trees and Uniform application

In order to define uniform recursion, our **third step** is to introduce an operation we call uniform application.

- 1 If $f : E \rightarrow D$ denotes a family of trees of D indexed on $e : E$, we call **future**(f) a future tree.
- 2 **future**(f) represents a tree whose root has an “unknown” label **future**, an “unknown” index set E , and children all $f(e)$ for any $e : E$.
- 3 We allow a unique way of acting on the tree **future**(f): to apply the same map g on all children $f(e)$ of **future**(f).
- 4 To this aim, we introduce a constant **u** we call “uniform application”, defined by **u**(**future**(f), g) = **future**($g \circ f$).
- 5 We **cannot act separately** on the children $g(f(e))$ of **future**($g \circ f$).
- 6 We extend recursive definition to future trees using *uniform application*.

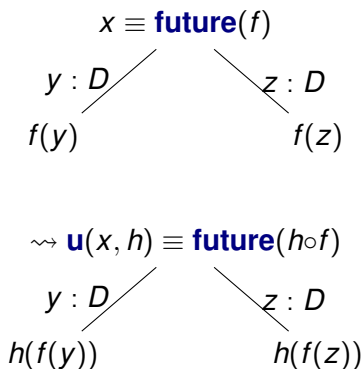
Future trees and Uniform application

In order to define uniform recursion, our **third step** is to introduce an operation we call uniform application.

- 1 If $f : E \rightarrow D$ denotes a family of trees of D indexed on $e : E$, we call **future**(f) a future tree.
- 2 **future**(f) represents a tree whose root has an “unknown” label **future**, an “unknown” index set E , and children all $f(e)$ for any $e : E$.
- 3 We allow a unique way of acting on the tree **future**(f): to apply the same map g on all children $f(e)$ of **future**(f).
- 4 To this aim, we introduce a constant **u** we call “uniform application”, defined by $\mathbf{u}(\mathbf{future}(f), g) = \mathbf{future}(g \circ f)$.
- 5 We **cannot act separately** on the children $g(f(e))$ of **future**($g \circ f$).
- 6 We extend recursive definition to future trees using *uniform application*.

Uniform application to a future tree

Uniform application applies a map $h : D \rightarrow D$ to all children of a future tree **future**(f).



Definition of Uniform Recursion

We have now all ingredients we need to define uniform recursion.

- 1 Uniform recursion on D in system \mathcal{N} defines a map $h : D \rightarrow A$ using one clause r_i for each D_i , and a single extra clause $r_n : A \rightarrow A$, dealing with all pending extensions $A@E$ of A through a uniform application.

- 2 If $\mathbf{t} \equiv \mathbf{c}_i(\mathbf{f})$, as usual we set

$$\mathbf{h}(\mathbf{t}) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$$

- 3 The clause r_n is used on trees $\mathbf{t} \equiv \mathbf{future}(\mathbf{f}) : \mathbf{D}$. We assume that $h : D \rightarrow A$ is already defined on $f(e) : D$ for all $e : D_i$, we *uniformly* apply h to the tree t forming $\mathbf{future}(h \circ \mathbf{f}) : A$, then we apply r_n obtaining

$$\mathbf{h}(\mathbf{t}) = \mathbf{r}_n(\mathbf{future}(\mathbf{h} \circ \mathbf{f}))$$

Definition of Uniform Recursion

We have now all ingredients we need to define uniform recursion.

- 1 Uniform recursion on D in system \mathcal{N} defines a map $h : D \rightarrow A$ using one clause r_i for each D_i , and a single extra clause $r_n : A \rightarrow A$, dealing with all pending extensions $A@E$ of A through a uniform application.

- 2 If $\mathbf{t} \equiv \mathbf{c}_i(\mathbf{f})$, as usual we set

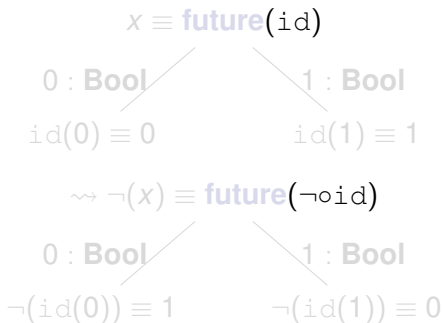
$$\mathbf{h}(\mathbf{t}) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$$

- 3 The clause r_n is used on trees $\mathbf{t} \equiv \mathbf{future}(\mathbf{f}) : \mathbf{D}$. We assume that $h : D \rightarrow A$ is already defined on $f(e) : D$ for all $e : D_i$, we *uniformly* apply h to the tree t forming $\mathbf{future}(h \circ \mathbf{f}) : A$, then we apply r_n obtaining

$$\mathbf{h}(\mathbf{t}) = \mathbf{r}_n(\mathbf{future}(\mathbf{h} \circ \mathbf{f}))$$

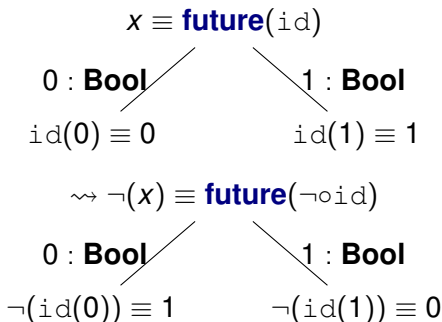
An example: one uniform extension of negation

- 1 Let **Bool** = {0, 1}. **Bool** with future constructors is a set of trees whose leaves are booleans.
- 2 Let $\neg(0) = 1$, $\neg(1) = 0$. We uniformly extend \neg to any future constructor of **Bool** with the clause $\neg(\mathbf{future}(f)) = \mathbf{future}(\neg \circ f)$.
- 3 The result is a map negating all leaves of a tree.



An example: one uniform extension of negation

- 1 Let **Bool** = {0, 1}. **Bool** with future constructors is a set of trees whose leaves are booleans.
- 2 Let $\neg(0) = 1, \neg(1) = 0$. We uniformly extend \neg to any future constructor of **Bool** with the clause $\neg(\mathbf{future}(f)) = \mathbf{future}(\neg \circ f)$.
- 3 The result is a map negating all leaves of a tree.



- §1. *Data types and Types of system \mathcal{N}*
- §2. *Primitive Recursion on Trees*
- §3. *Recursion in system \mathcal{N} : Uniform Recursion*
- **§4. Definitions and Results for System \mathcal{N}**
- §5. *Conclusions*

§4. Definitions and Results for System \mathcal{N}

We mark the new parts in **red**.

Definition (Terms of \mathcal{N})

Let $n, m \in \mathbf{Nat}$, $i < n, j < m$, $D = (D_0, \dots, D_{n-1})$, $E \in \mathbf{Data}$, $A, B \in \mathbf{Tp}$, and $\Gamma = E_0, \dots, E_{m-1}$ any context :

- 1 $\Gamma \vdash C : A$: if $C(\vec{x}) = \alpha[\vec{x}]$ is a combinator of type A
- 2 $\Gamma \vdash \langle -, - \rangle : A, B \rightarrow A_1 \times A_2$ and $\Gamma \vdash \pi_i : A_1 \times A_2 \rightarrow A_i$
- 3 (constructors) $\Gamma \vdash \mathbf{cons}_{i,D} : (D_i \rightarrow D) \rightarrow D$
- 4 If $\Gamma \vdash t : A \rightarrow B$ and $\Gamma \vdash u : A$, then $\Gamma \vdash t(u) : B$.
- 5 (**future constructors**) $\Gamma \vdash \mathbf{future}_{j,E_j,A} : (E_j \rightarrow A) \rightarrow A$
- 6 (**uniform application**) $\Gamma \vdash \mathbf{u}_D : D, (D \rightarrow D) \rightarrow D$
- 7 (**recursion**) $\Gamma \vdash \mathbf{r}_{D,A} : \vec{R}, D \rightarrow A$, with $R_i = (D_i \rightarrow A) \rightarrow A$ for all $i < n$, and $R_n = (A \rightarrow A)$
- 8 (**Forth**) If $\Gamma, E, \Delta \vdash t : A$, then $\Gamma, \Delta \vdash \mathbf{Forth}_{m,E}.t : A@E$

Lifting application

We explained how to apply a map $h : D \rightarrow A$ to a larger domain $D@E$: we call this operation “Lifting”. We may internalize lifting in \mathcal{N} through a term **Lift** : $(D \rightarrow A) \rightarrow (D@E \rightarrow A@E)$

- 1 We first define a recursive map **Back** : $D@E \rightarrow D$, turning the last constructor **c** : $(E \rightarrow D@E) \rightarrow D@E$ of $D@E$ into a future constructor **future** _{m,E} : $(E \rightarrow D) \rightarrow D$ of D
- 2 Then we apply h to the result **Back**(t), obtaining some $h(\mathbf{Back}(t)) : A$
- 3 Eventually we use the **Forth** operator, replacing **future** with **c**: the result has type $A@E$.
- 4 Let $C(x, y, z) \equiv y(x(z))$: then we define

$$\mathbf{Lift} \equiv \mathbf{Forth}(C(\mathbf{Back}))$$

Lifting application

We explained how to apply a map $h : D \rightarrow A$ to a larger domain $D@E$: we call this operation “Lifting”. We may internalize lifting in \mathcal{N} through a term **Lift** : $(D \rightarrow A) \rightarrow (D@E \rightarrow A@E)$

- 1 We first define a recursive map **Back** : $D@E \rightarrow D$, turning the last constructor **c** : $(E \rightarrow D@E) \rightarrow D@E$ of $D@E$ into a future constructor **future** _{m,E} : $(E \rightarrow D) \rightarrow D$ of D
- 2 Then we apply h to the result **Back**(t), obtaining some $h(\mathbf{Back}(t)) : A$
- 3 Eventually we use the **Forth** operator, replacing **future** with **c**: the result has type $A@E$.
- 4 Let $C(x, y, z) \equiv y(x(z))$: then we define

$$\mathbf{Lift} \equiv \mathbf{Forth}(C(\mathbf{Back}))$$

The role of Lifting in system \mathcal{N}

- 1 The only limitation we have for lifting an application from D to $D@E$ is that h may act on any tree $c(f)$ defined by the constructor $c : (E \rightarrow D@E) \rightarrow D@E$ only using *uniform application*.
- 2 h cannot select a child $f(e)$ of $c(f)$ for some $e : E$, because **we assume that h does not know the type E .**
- 3 We believe that this limitation is essential in order insure termination and to match exactly the expressive power of polymorphism of system \mathcal{F} .

The role of Lifting in system \mathcal{N}

- 1 The only limitation we have for lifting an application from D to $D@E$ is that h may act on any tree $c(f)$ defined by the constructor $c : (E \rightarrow D@E) \rightarrow D@E$ only using *uniform application*.
- 2 h cannot select a child $f(e)$ of $c(f)$ for some $e : E$, because **we assume that h does not know the type E** .
- 3 We believe that this limitation is essential in order insure termination and to match exactly the expressive power of polymorphism of system \mathcal{F} .

The normalization and the well-foundedness results

- 1 The main feature of \mathcal{N} is that we combine the fact that the domain of a map is extendable with the fact that all maps are total.
- 2 Indeed, using the notion of approximation (*see Appendix*) and Tait's notion of reducibility we may intuitionistically prove:

Theorem (Normalization and Well-foundedness)

- 1 *All terms of \mathcal{N} normalize*
- 2 *all trees denoted by some term $t : D \in \mathbf{Data}$ of system \mathcal{N} are **well-founded**.*

The normalization and the well-foundedness results

- 1 The main feature of \mathcal{N} is that we combine the fact that the domain of a map is extendable with the fact that all maps are total.
- 2 Indeed, using the notion of approximation (*see Appendix*) and Tait's notion of reducibility we may intuitionistically prove:

Theorem (Normalization and Well-foundedness)

- 1 *All terms of \mathcal{N} normalize*
- 2 *all trees denoted by some term $t : D \in \mathbf{Data}$ of system \mathcal{N} are **well-founded**.*

§5: Conclusions

- 1 **We defined a simply typed λ -calculus \mathcal{N}** in which primitive recursive definitions on trees may be extended to a larger domain using uniform application
- 2 System \mathcal{N} is defined in term of **concrete tree operations** and aims to be **equivalent to polymorphism**.
- 3 **What we proved**: System \mathcal{N} has the usual properties of Subject Reduction, Confluence and Normalization
- 4 **What we are checking**: whether system \mathcal{N} defines a Realization Model for Second Order Arithmetic, and whether the definable maps on data types are the same in system \mathcal{N} and system \mathcal{F} .

§5: Conclusions

- 1 **We defined a simply typed λ -calculus \mathcal{N}** in which primitive recursive definitions on trees may be extended to a larger domain using uniform application
- 2 System \mathcal{N} is defined in term of **concrete tree operations** and aims to be **equivalent to polymorphism**.
- 3 **What we proved**: System \mathcal{N} has the usual properties of Subject Reduction, Confluence and Normalization
- 4 **What we are checking**: whether system \mathcal{N} defines a Realization Model for Second Order Arithmetic, and whether the definable maps on data types are the same in system \mathcal{N} and system \mathcal{F} .

§5: Conclusions

- 1 **We defined a simply typed λ -calculus \mathcal{N}** in which primitive recursive definitions on trees may be extended to a larger domain using uniform application
- 2 System \mathcal{N} is defined in term of **concrete tree operations** and aims to be **equivalent to polymorphism**.
- 3 **What we proved**: System \mathcal{N} has the usual properties of Subject Reduction, Confluence and Normalization
- 4 **What we are checking**: whether system \mathcal{N} defines a Realization Model for Second Order Arithmetic, and whether the definable maps on data types are the same in system \mathcal{N} and system \mathcal{F} .

- 1 P. Martin-Lof, Intuitionistic Type Theory, June 1980, Bibliopolis.
- 2 H. Barendregt, Lambda Calculus with Types. Cambridge University Press, 2013.
- 3 William W. Tait: Intensional Interpretations of Functionals of Finite Type I. J. Symb. Log. 32(2): 198-212 (1967)

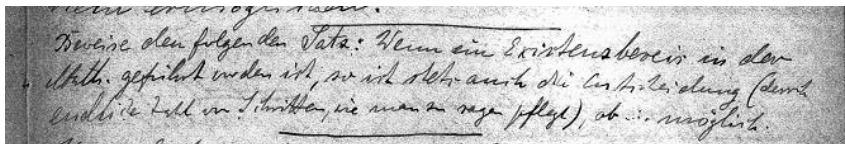


Figure : Hilbert Constructivization Conjecture (Courtesy from Goettingen State and University Library, Germany. Thanks to Benedikt Ahrens for translating).

Probably the first version (around 1917) of the following conjecture by Hilbert:

”Prove the following theorem: When a proof of existence has been concluded in mathematics, then also the decision (in a finite number of steps, as one says) is always possible. ”

Appendix: the system \mathcal{N}

- 1 Terms of \mathcal{N} of type A are defined w.r.t. a list $\Gamma \equiv E_0, \dots, E_{m-1}$ of data structures, denoting *pending extensions* of A .
- 2 Terms of \mathcal{N} include all algebraic combinators, pairing, projections, tree constructors, future constructors
future : $(E \rightarrow A) \rightarrow A$, uniform application
u : $E, (E \rightarrow E) \rightarrow E$ and for each $D = (D_0, \dots, D_{n-1})$ a constant **r** $\equiv \mathbf{r}_{D,A}$ denoting recursion on trees of D with result in A .
- 3 **r** has one recursive clause $r_i : (D_i \rightarrow A) \rightarrow A$ for each index set D_i , and one extra clause $r_n : A \rightarrow A$, dealing with extensions of A .
- 4 Terms are closed under the unary operator **Forth** $_{m,E}$, which removes the type E in position m from a context, and under application.
- 5 We write $\Gamma \vdash t : A$ for “ $t : A$ in the context Γ ”.

Definition (Algebraic Reductions for \mathcal{N})

- 1 Let $C(\vec{x}) = \alpha[\vec{t}]$ be any combinator.
 - 1 $C(\vec{t}) \rightsquigarrow \alpha[\vec{t}/\vec{x}]$.
 - 2 $\pi_i(\langle a_1, a_2 \rangle) \rightsquigarrow a_i$ for $i = 1, 2$
 - 3 If $a \rightsquigarrow b$ then $\pi_i(a) \rightsquigarrow \pi_i(b)$.
 - 4 If $f \rightsquigarrow g$ then $fa \rightsquigarrow ga$
- 2 Let $c \equiv \mathbf{future}_{i,E}$ and P be the combinator postponing an application, defined by $P(x, y) = y(x)$
 - 1 $c(f)(e) \rightsquigarrow c(Pe \circ f)$
 - 2 $\pi_i(c(f)) \rightsquigarrow c(\pi_i \circ f)$ for $i = 1, 2$

Definition (Uniform Reductions for \mathcal{N})

- 1 Let $c \equiv \mathbf{future}_{i,E,D}$, $\mathbf{cons}_{i,D}$ and $g : D \rightarrow D$ and $c(f) : D$.
 - 1 $\mathbf{u}(c(f))(g) \rightsquigarrow c(g \circ f) : B$
 - 2 If $d \rightsquigarrow e : D$ then $\mathbf{u}(d)(g) \rightsquigarrow \mathbf{u}(e)(g)$
- 2 Assume $D = (D_0, \dots, D_{n-1})$, $\vec{r} = r_0, \dots, r_n$.
 - 1 If $d \equiv c(f)$ and $c \equiv \mathbf{cons}_{i,D_i}$ for some $i < n$ then $\mathbf{r}\vec{r}d \rightsquigarrow r_i(\mathbf{r}(\vec{r}) \circ f)$
 - 2 If $d \equiv c(f)$ and $c \equiv \mathbf{future}$ then $\mathbf{r}\vec{r}d \rightsquigarrow r_n(c(\mathbf{r}(\vec{r}) \circ f))$
 - 3 If $d \rightsquigarrow e$ then $\mathbf{r}\vec{r}d \rightsquigarrow \mathbf{r}\vec{r}e$

If $\Gamma, \Delta \vdash a : A$ then we denote with $a^{i,E}$ the term corresponding of a in the context Γ, E, Δ (defined in the next slide).

Definition (Reductions for Forth)

Assume $D = (D_0, \dots, D_{n-1})$.

- 1 **Forth**(future $_{i,E,D}f$) \rightsquigarrow **C** $_{n,D@E}$ (**Forth**(f))
- 2 **Forth**(future $_{j+1,E,D}f$) \rightsquigarrow future $_{j,E,D@E}$ (**Forth**(f)) for $j \geq i$
- 3 **Forth**(cf) \rightsquigarrow **c**(**Forth**(f)) for any other constructor or future constructor.
- 4 If $d : D \in$ **Data** and $d \rightsquigarrow e : D$ then **Forth** $d \rightsquigarrow$ **Forth** e .
- 5 **Forth**(f)(a) \rightsquigarrow **Forth**($f(a^{i,E})$)
- 6 π_i (**Forth**(a)) \rightsquigarrow **Forth**($\pi_i(a)$) for $i = 1, 2$.

Context Lifting

Context Lifting is an operation moving a term t in the context Γ, Δ to the corresponding term $t^{i,E}$ in the context Γ, E, Δ .
Context lifting adds 1 to the subscripts of future constructors with index in Δ .

Definition (The term $t^{i,E}$)

Assume $\Gamma \vdash t : A$ is a term of \mathcal{N} , c is any constant. We define $t^{i,E}$ by induction on t .

- 1 **future** $_{j,F}$ $^{i,E} \equiv \mathbf{future}_{j+1,F}$ for all $j \geq i$
- 2 $c^{i,E} \equiv c$ in all other cases.
- 3 **Forth** $_{j,F}(u)^{i,E} \equiv \mathbf{Forth}_{j+1,F}(u^{i,E})$ for all $j \geq i$.
- 4 **Forth** $_{j,F}(u)^{i,E} \equiv \mathbf{Forth}_{j,F}(u^{j+1,E})$ in all other cases.
- 5 $t(u)^{i,E} \equiv t^{i,E}(u^{i,E})$

Appendix: Models of system \mathcal{N} and a vicious cycle

In order to define a model of system \mathcal{N} we face the following vicious cycle.

- 1 The definition of a term $E \vdash t : D$ may include a constructor c of index set D .
- 2 c has type $: (E \rightarrow D) \rightarrow D$, and has domain all maps $E \rightarrow D$.
- 3 If $E = D$, defining these maps requires to define D **before completing the definition** of any $t : D$.

In order to break this vicious cycle, we have to exploit the fact that a map acts on a tree $c(f)$ by uniform application only, without knowing the type E .

Approximated constructors and a vicious cycle

Let $E \in \mathbf{Data}$, and \mathcal{A} be any model of \mathcal{N} , and $E_{\mathcal{A}}$ the set of elements of empty context and type E in \mathcal{A} and $X \subseteq E_{\mathcal{A}}$.

- 1 In the models of \mathcal{N} we add constants $j_X : (X \rightarrow D) \rightarrow D$ we call *approximated constructors*.
- 2 We use j_X to represent a constructor $c : (E \rightarrow D) \rightarrow D$.
- 3 If the constant j_X is in \mathcal{A} , then $X \subset E_{\mathcal{A}}$.
- 4 Indeed, the terms of empty context and type E defined from j_X are in $E_{\mathcal{A}}$ and not in X .
- 5 In every single model \mathcal{A} , we found for approximated constructors a vicious cycle similar to the vicious cycle for constructor.
- 6 This second vicious cycle, however, is easier to break.

Appendix: Breaking the vicious cycle

- 1 For any model \mathcal{A} there is a model $\mathcal{B} \supset \mathcal{A}$ including the approximated constructor $\mathfrak{j}_{E_{\mathcal{A}}}$.
- 2 Therefore the behavior of the family of terms $c_{\mathcal{A}}$ in context $\Gamma = E$ for all \mathcal{A} may be described from the behavior of the family of terms $\mathfrak{j}_{E_{\mathcal{A}}}$ in the empty context for all \mathcal{A} .
- 3 By exploiting this idea we may adapt Tait's notion of reducibility ([3]) to system \mathcal{N} .
- 4 We express Tait's reducibility w.r.t. all models of \mathcal{N} and not just w.r.t. a single model of \mathcal{N} .