

The Simply Typed Lambda Calculus \mathcal{N}

EUTypes Meeting in Nijmegen

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

Torino University, Italy

January 23, 2018



A First Order Reformulation of Polymorphism

- 1 We introduce a simply typed λ -calculus, **system** \mathcal{N}
- 2 Our goal is representing in \mathcal{N} all well-founded trees and polymorphic maps on them which are definable in **system** \mathcal{F}
- 3 The main feature of \mathcal{N} is the possibility of extending at run-time the domain of a recursively defined map

A First Order Reformulation of Polymorphism

- 1 We introduce a simply typed λ -calculus, **system \mathcal{N}**
- 2 Our goal is representing in \mathcal{N} all well-founded trees and polymorphic maps on them which are definable in **system \mathcal{F}**
- 3 The main feature of \mathcal{N} is the possibility of extending at run-time the domain of a recursively defined map
- 4 Reduction of \mathcal{N} are:
 - **algebraic** reductions
 - reductions for **primitive recursion** on trees
 - reductions for **adding one constructor** to a recursive definition

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**.
- 3 system \mathcal{N} defines a **Infinitary Proof System** for second order intuitionistic arithmetic


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**.
- 3 system \mathcal{N} defines a **Infinitary Proof System** for second order intuitionistic arithmetic

The results we are checking is:

- 1 **equivalence between system \mathcal{N} and polymorphism**
- 2 There is a fully-abstract model of system \mathcal{N}

- §1. Types of \mathcal{N} 
- §2. Recursion in \mathcal{N}
- §3. Terms of \mathcal{N}
- §4. Semantics for Expandable Recursion
- §5. Conclusions

§1. Types of system \mathcal{N}

- 1 Let (D_1, \dots, D_n) be the set of well-founded trees whose constructors have index sets among D_1, \dots, D_n .
- 2 In Martin-Lof notation [1], (D_1, \dots, D_n) is the W -type $W(i : \{1, \dots, n\})D_i$.

§1. Types of system \mathcal{N}

- 1 Let (D_1, \dots, D_n) be the set of well-founded trees whose constructors have index sets among D_1, \dots, D_n .
- 2 In Martin-Lof notation [1], (D_1, \dots, D_n) is the W -type $W(i : \{1, \dots, n\})D_i$.
- 3 The set **Data** of data types of \mathcal{N} is the smallest set such that: if $D_1, \dots, D_n \in \mathcal{D}$ then $(D_1, \dots, D_n) \in \mathcal{D}$.
- 4 **Data** includes the data types: \emptyset , **Unit**, **Bool**, **Nat**, *finite binary trees*, *well-founded at most countable trees*, and many more.

§1. Types of system \mathcal{N}

- 1 Let (D_1, \dots, D_n) be the set of well-founded trees whose constructors have index sets among D_1, \dots, D_n .
- 2 In Martin-Lof notation [1], (D_1, \dots, D_n) is the W -type $W(i : \{1, \dots, n\})D_i$.
- 3 The set **Data** of data types of \mathcal{N} is the smallest set such that: if $D_1, \dots, D_n \in \mathcal{D}$ then $(D_1, \dots, D_n) \in \mathcal{D}$.
- 4 **Data** includes the data types: \emptyset , **Unit**, **Bool**, **Nat**, *finite binary trees*, *well-founded at most countable trees*, and many more.
- 5 The set **Tp** of types of \mathcal{N} is inductively defined by $T ::= D \mid T \times T \mid T \rightarrow T$ for any data type $D \in \mathbf{Data}$.

The general pattern for a Tree

$D = (D', D'', D''')$ is the set of well-founded trees whose nodes have index set D' or D'' or D''' . A tree in D is made of:

- 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 $y, z : D', t, u : D'', v, w : D'''$

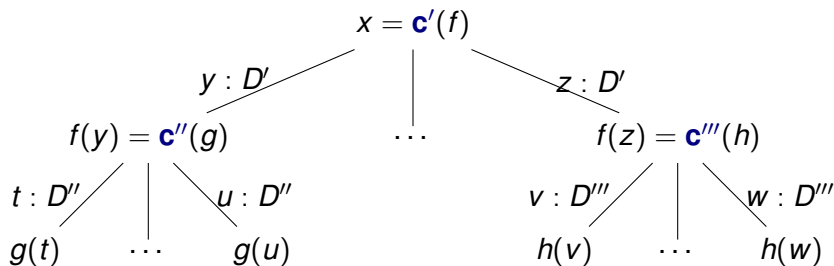
Assume that $f(y), f(y)$ have values $c''(g), c'''(h)$. Then $x = c'(f) : D$ denotes the tree:


The general pattern for a Tree

$D = (D', D'', D''')$ is the set of well-founded trees whose nodes have index set D' or D'' or D''' . A tree in D is made of:

- 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. Types of \mathcal{N}
- §2. Recursion in \mathcal{N} 
- §3. Terms of \mathcal{N}
- §4. Semantics for Expandable Recursion
- §5. Conclusions

§2. Recursion in \mathcal{N}

- 1 Informally, a primitive recursion of $h : D \rightarrow A$ runs as follows. 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

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

The constructor \mathbf{c}_i disappears.

§2. Recursion in \mathcal{N}

- 1 Informally, a primitive recursion of $h : D \rightarrow A$ runs as follows. 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

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

The constructor \mathbf{c}_i disappears.

- 2 In any $D \in \mathbf{Data}$, the list d_1, \dots, d_n, \dots of arguments of \mathbf{c}_i is expressed in \mathcal{N} by an index map $f : D_i \rightarrow D$ such that $f(e_1) = d_1, \dots, f(e_n) = d_n, \dots$ for some $e_1, \dots, e_n, \dots : D_i$.

§2. Recursion in \mathcal{N}

- 1 Informally, a primitive recursion of $h : D \rightarrow A$ runs as follows. 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

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

The constructor \mathbf{c}_i disappears.

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

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

Definition of Extendable Recursion

- 1 We define a map $h : D \rightarrow A$ using one clause r_i for each constructor \mathbf{c}_i , and a **single extra clause** $r_n : A \rightarrow A$, dealing with all possible extensions of the data type D .
- 2 If $\mathbf{t} \equiv \mathbf{c}_i(\mathbf{f})$, then $\mathbf{h}(\mathbf{t}) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$

Definition of Extendable Recursion

- 1 We define a map $h : D \rightarrow A$ using one clause r_i for each constructor \mathbf{c}_i , and a **single extra clause** $r_n : A \rightarrow A$, dealing with all possible extensions of the data type D .
- 2 If $\mathbf{t} \equiv \mathbf{c}_i(\mathbf{f})$, then $\mathbf{h}(\mathbf{t}) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$
- 3 The clause r_n is used on trees $\mathbf{t} \equiv \mathbf{future}(\mathbf{f}) : \mathbf{D}$ built by some new constructor **future**. We assume that $h : D \rightarrow A$ is already defined on $f(e) : D$ for all $e : D_i$, we move the recursive call to h to the children of t , forming **future**($h \circ f$) : A , then we apply r_n obtaining

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

The constructor **future** does **not** disappear.

Definition of Extendable Recursion

- 1 We define a map $h : D \rightarrow A$ using one clause r_i for each constructor \mathbf{c}_i , and a **single extra clause** $r_n : A \rightarrow A$, dealing with all possible extensions of the data type D .
- 2 If $\mathbf{t} \equiv \mathbf{c}_i(\mathbf{f})$, then $\mathbf{h}(\mathbf{t}) = \mathbf{r}_i(\mathbf{h} \circ \mathbf{f})$
- 3 The clause r_n is used on trees $\mathbf{t} \equiv \mathbf{future}(\mathbf{f}) : \mathbf{D}$ built by some new constructor **future**. We assume that $h : D \rightarrow A$ is already defined on $f(e) : D$ for all $e : D_i$, we move the recursive call to h to the children of t , forming **future**($h \circ f$) : A , then we apply r_n obtaining

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

The constructor **future** does **not** disappear.

- 4 This way of computing with a “generic” constructor is different from all ways we know of, for instance from **Jay’s reduction** for a generic binary constructor ([3], p.101).

Recursion with Extendable Domain in \mathcal{N}

In order to extend the domain of a recursive map at run-time, 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 type $D = (D_0, \dots, D_{n-1})$:
 $D@E = (D_0, \dots, D_{n-1}, E) \in \mathbf{Data}$.
- 2 $D@E$ has one tree constructor $\mathbf{c}_n : (E \rightarrow D@E) \rightarrow D@E$ more than D .

Recursion with Extendable Domain in \mathcal{N}

In order to extend the domain of a recursive map at run-time, 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 type $D = (D_0, \dots, D_{n-1})$:
 $D@E = (D_0, \dots, D_{n-1}, E) \in \mathbf{Data}$.
- 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 extend the domain of a recursive map at run-time, our **second step** is to introduce constants

future $_{m,E,D} : (E \rightarrow D) \rightarrow D$ we call **future constructors**.

- 1 If $D = (D_0, \dots, D_{n-1})$, then **future** $_{m,E,D}$ represents in D a constructor **c_n** we may add to D in a possible extension $D @ E = (D_0, \dots, D_{n-1}, E)$.

Future constructors

In order to extend the domain of a recursive map at run-time, our **second step** is to introduce constants

future $_{m,E,D} : (E \rightarrow D) \rightarrow D$ we call **future constructors**.

- 1 If $D = (D_0, \dots, D_{n-1})$, then **future** $_{m,E,D}$ represents in D a constructor **c_n** we may add to D in a possible extension $D@E = (D_0, \dots, D_{n-1}, E)$.
- 2 We add a unary **term operator** **Forth** $_{m,E}$, executing a possible extension of D with E .

Future constructors

In order to extend the domain of a recursive map at run-time, our **second step** is to introduce constants

future $_{m,E,D} : (E \rightarrow D) \rightarrow D$ we call **future constructors**.

- 1 If $D = (D_0, \dots, D_{n-1})$, then **future** $_{m,E,D}$ represents in D a constructor \mathbf{c}_n we may use to extend D in a possible extension $D@E = (D_0, \dots, D_{n-1}, E)$.
- 2 We add a unary **term operator** **Forth** $_{m,E}$, executing a possible extension of D with E .
- 3 **Forth** $_{m,E}$ replaces **future** $_{m,E,D}$ with \mathbf{c}_n :

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

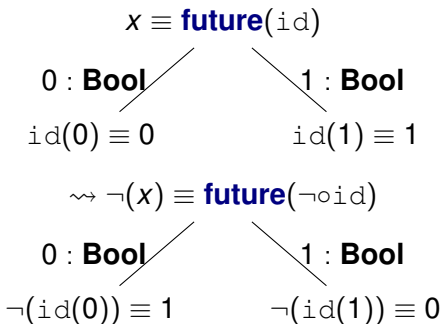
- 4 **future** $_{m,E,D}$, **Forth** $_{m,E}$ are extended to all types point-wise and component-wise.


An example: one uniform extension of negation

- 1 Let $\mathbf{Bool} = \{0, 1\}$. \mathbf{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 \mathbf{Bool} with the clause $\neg(\mathbf{future}(f)) = \mathbf{future}(\neg \circ f)$.

An example: one uniform extension of negation

- 1 Let $\mathbf{Bool} = \{0, 1\}$. \mathbf{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 \mathbf{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. Types of \mathcal{N}
- §2. Recursion in \mathcal{N}
- §3. Terms of \mathcal{N} 
- §4. Semantics for Expandable Recursion
- §5. Conclusions

§3. Terms of \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 types, denoting *possible extensions* of A .
- 2 Terms of \mathcal{N} include all algebraic combinators, pairing, projections, tree constructors, future constructors **future** : $(E \rightarrow A) \rightarrow A$ for $E \in \Gamma$, uniform application **u** : $D, (D \rightarrow D) \rightarrow D$ 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 application, and under the unary operator **Forth** $_{m,E}$, which removes the type E in position m from a context.
- 5 We write $\Gamma \vdash t : A$ for “ $t : A$ in the context Γ ”.

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 $\Gamma \vdash \pi_j : A_1 \times A_2 \rightarrow A_j$
- 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 $\Gamma \vdash t(u) : B$.

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 $\Gamma, \Delta \vdash \mathbf{Forth}_{m,E}.t : A@E$

Definition (Reductions on \mathbf{u} , \mathbf{r} for \mathcal{N})

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

The remaining reductions for \mathcal{N} are given in Appendix.

- §1. Types of \mathcal{N}
- §2. Recursion in \mathcal{N}
- §3. Terms of \mathcal{N}
- §4. Semantics for Expandable Recursion 
- §5. Conclusions

§4. Semantics for Expandable Recursion

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 future constructor **future** _{E} of index set E .
- 2 **future** _{E} 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 $t : D$.
- 4 Thus, the definition of **future** _{E} is not stratified.

Candidates and Approximated Constructors

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

- 1 We call X a **candidate** for $E_{\mathcal{A}}$.
- 2 In the models of \mathcal{N} we add constants $j_X : (X \rightarrow D) \rightarrow D$.
- 3 We call j_X an **approximation** of the future constructor **future** $_E : (E \rightarrow D) \rightarrow D$.

Candidates and Approximated Constructors

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

- 1 We call X a **candidate** for $E_{\mathcal{A}}$.
- 2 In the models of \mathcal{N} we add constants $j_X : (X \rightarrow D) \rightarrow D$.
- 3 We call j_X an **approximation** of the future constructor **future** $_E : (E \rightarrow D) \rightarrow D$.
- 4 The branching of $j_X(f)$ is a **restriction** of the branching of **future** $_E(f)$.
- 5 The definition of j_X is stratified, therefore if we may interpret **future** $_E = j_X$ we would be done.

Unfortunately ... (see next slide)

A second vicious cycle

- 1 Unfortunately, we cannot have $\mathbf{future}_E = j_X$ in \mathcal{A} .
- 2 Indeed, if we choose $X \subseteq E_{\mathcal{A}}$ and we add the new constant j_X to \mathcal{A} , then we may define new terms $e \in E_{\mathcal{A}}$ from them.
- 3 e is defined after X , thus we may have $e \notin X$, hence $X \neq E$ and $\mathbf{future}_E \neq j_X$.

A second vicious cycle


- 1 Unfortunately, we cannot have **future** $_E = j_X$ in \mathcal{A} .
- 2 Indeed, if we choose $X \subseteq E_{\mathcal{A}}$ and we add the new constant j_X to \mathcal{A} , then we may define new terms $e \in E_{\mathcal{A}}$ from them.
- 3 e is defined after X , thus we may have $e \notin X$, hence $X \neq E$ and **future** $_E \neq j_X$.
- 4 If we try to **force** $X = E_{\mathcal{A}}$ in \mathcal{A} , we find a vicious cycle similar to the vicious cycle in the definition of constructor.
- 5 This second vicious cycle, however, is easier to break.

Breaking the vicious cycle

- 1 For any model \mathcal{A} there is a model $\mathcal{A}^E \supset \mathcal{A}$ including the approximated constructor $\mathfrak{j}_{E_{\mathcal{A}}}$.
- 2 \mathcal{N} cannot distinguish between **future**_E and $\mathfrak{j}_{E_{\mathcal{A}}}$: thus, the behavior of **future**_E in \mathcal{A} may be described from the behavior of $\mathfrak{j}_{E_{\mathcal{A}}}$ in \mathcal{A}^E , without any vicious cycle.

Breaking the vicious cycle

- 1 For any model \mathcal{A} there is a model $\mathcal{A}^E \supset \mathcal{A}$ including the approximated constructor $\mathfrak{j}_{E_{\mathcal{A}}}$.
- 2 \mathcal{N} cannot distinguish between **future**_E and $\mathfrak{j}_{E_{\mathcal{A}}}$: thus, the behavior of **future**_E in \mathcal{A} may be described from the behavior of $\mathfrak{j}_{E_{\mathcal{A}}}$ in \mathcal{A}^E , without any vicious cycle.
- 3 By exploiting this idea we may adapt Tait's notion of reducibility ([4]) to system \mathcal{N} .
- 4 We express Tait's reducibility w.r.t. a countable family of models of \mathcal{N} , closed under the operation $\mathcal{A} \mapsto \mathcal{A}^E$.
- 5 This proof cannot be expressed in a **second order arithmetic**, unless **we bound** the number of nesting in a data type and in a type.

- §1. Types of \mathcal{N}
- §2. Recursion in \mathcal{N}
- §3. Terms of \mathcal{N}
- §4. Semantics for Expandable Recursion
- §5. Conclusions 

§5: Conclusions

The main feature of \mathcal{N} is: the domain of a map of \mathcal{N} is extendable at run-time, yet all maps are total.

§5: Conclusions

The main feature of \mathcal{N} is: the domain of a map of \mathcal{N} is extendable at run-time, yet all maps are total.

Theorem (Totality and Expressive Power of \mathcal{N})

- 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**.
- 3 (Expressive Power) We may define in \mathcal{N} an **Infinitary Proof System** for second order intuitionistic arithmetic \mathbf{HA}^2

Summary of the Talk

- 1 **We defined a simply typed λ -calculus \mathcal{N}** in which primitive recursive definitions on trees may be extended to a larger domain at run-time.
- 2 System \mathcal{N} is defined in term of **concrete tree operations** and aims to be **equivalent to polymorphism**.

Summary of the Talk

- 1 **We defined a simply typed λ -calculus \mathcal{N}** in which primitive recursive definitions on trees may be extended to a larger domain at run-time.
- 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, and defines a Infinitary Proof System for Second Order Arithmetic.

Summary of the Talk

- 1 **We defined a simply typed λ -calculus \mathcal{N}** in which primitive recursive definitions on trees may be extended to a larger domain at run-time.
- 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, and defines a Infinitary Proof System for Second Order Arithmetic.
- 4 **What we are checking:** whether well-founded trees and the definable maps on them are the same in system \mathcal{N} and system \mathcal{F} , and whether \mathcal{N} defines a denotation system for ordinals of second order analysis.

References

- 1 P. Martin-Lof, Intuitionistic Type Theory, June 1980, Bibliopolis.
- 2 H. Barendregt, Lambda Calculus with Types. Cambridge University Press, 2013.
- 3 B. Jay, D. Kesner, Pure pattern calculus, ESOP'06, Vienna, p. 100-114.
- 4 William W. Tait: Intensional Interpretations of Functionals of Finite Type I. J. Symb. Log. 32(2): 198-212 (1967)

A research report about system \mathcal{N} may be found at:

**`www.di.unito.it/~stefano/
SistemaN-definizioni-14-Luglio-2017.pdf`**

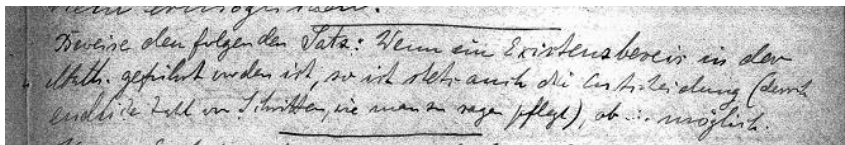


Figure: Hilbert Constructivization Conjecture (Courtesy from Goettingen State and University Library, Germany. Thanks to Benedikt Ahrens for translating).

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. ”

Definition (Algebraic Reductions for \mathcal{N})

- 1 Let $C(\vec{x}) = \alpha[\vec{x}]$ 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 (Reductions on \mathbf{u} , \mathbf{r} for \mathcal{N})

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

- 1 **Forth** upgrades a term from the context Γ, E, Δ to the context Γ, Δ , executing the extension of index set E .
- 2 **Forth** requires the operation $a^{i,E}$ of context lifting (*defined in the next slide*).

Definition (Reductions for **Forth**)

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

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

Context Lifting $t^{i,E}$

- 1 Context Lifting downgrades a term from the context Γ, Δ to the context Γ, E, Δ , adding the extension of index set E to the list of future extensions.
- 2 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 **(down-grading)** $(\mathbf{future}_{j,F})^{i,E} \equiv \mathbf{future}_{j+1,F}$ for all $j \geq i$
- 2 $c^{i,E} \equiv c$ in all other cases.
- 3 $\mathbf{Forth}_{j,F}(u)^{i,E} \equiv \mathbf{Forth}_{j+1,F}(u^{i,E})$ for all $j \geq i$.
- 4 $\mathbf{Forth}_{j,F}(u)^{i,E} \equiv \mathbf{Forth}_{j,F}(u^{j+1,E})$ in all other cases.
- 5 $t(u)^{i,E} \equiv t^{i,E}(u^{i,E})$