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 $\mathcal{N}$

Plan of the Talk

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

§1. Types of system $\mathcal{N}$

1. Let $(D_1, \ldots, D_n)$ be the set of well-founded trees whose constructors have index sets among $D_1, \ldots, D_n$.
2. In Martin-Lof notation [1], $(D_1, \ldots, D_n)$ is the W-type $W(i : \{1, \ldots, n\})D_i$.
3. The set **Data** of data types of $\mathcal{N}$ is the smallest set such that: if $D_1, \ldots, D_n \in \mathcal{D}$ then $(D_1, \ldots, D_n) \in \mathcal{D}$.
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 $(D_1, \ldots, D_n)$ be the set of well-founded trees whose constructors have index sets among $D_1, \ldots, D_n$.
2. In Martin-Lof notation [1], $(D_1, \ldots, D_n)$ is the W-type $W(i : \{1, \ldots, n\})D_i$.
3. The set **Data** of data types of $\mathcal{N}$ is the smallest set such that: if $D_1, \ldots, D_n \in \mathcal{D}$ then $(D_1, \ldots, D_n) \in \mathcal{D}$.
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 $T ::= D|T \times T|T \rightarrow T$ for any data type $D \in$ **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 $y, z : D', \quad t, u : D'', \quad v, w : D'''$

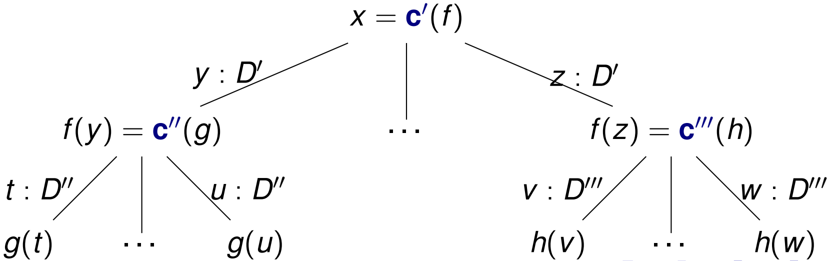
Assume that $f(y)$, $f(y)$ have values $c''(g)$, $c'''(h)$. Then $x = \mathbf{c}'(f) : D$ denotes the tree:

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 $D' \to D$, $D'' \to D$, $D''' \to D$
3. indexes $y, z : D'$, $\quad t, u : D''$, $\quad v, w : D'''$

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



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

1. Informally, a primitive recursion of $h : D \to A$ runs as follows. Assume that D has a constructor $\mathbf{c_i}$ with argument list $d_1, \ldots, d_n, \ldots$. We first apply h to each $d_1, \ldots, d_n, \ldots$, obtaining $h(d_1), \ldots, h(d_n), \ldots : A$, then we define

$$\mathbf{h}(\mathbf{c_i}(\mathbf{d_1}, \ldots, \mathbf{d_n}, \ldots)) = \mathbf{r_i}(\mathbf{h}(\mathbf{d_1}), \ldots, \mathbf{h}(\mathbf{d_n}), \ldots)$$

The constructor $\mathbf{c_i}$ disappear.

1. Informally, a primitive recursion of $h : D \to A$ runs as follows. Assume that D has a constructor $\mathbf{c_i}$ with argument list $d_1, \ldots, d_n, \ldots$. We first apply h to each $d_1, \ldots, d_n, \ldots$, obtaining $h(d_1), \ldots, h(d_n), \ldots : A$, then we define

$$\mathbf{h}(\mathbf{c_i}(\mathbf{d_1}, \ldots, \mathbf{d_n}, \ldots)) = \mathbf{r_i}(\mathbf{h}(\mathbf{d_1}), \ldots, \mathbf{h}(\mathbf{d_n}), \ldots)$$

   The constructor $\mathbf{c_i}$ disappear.

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

§2. Recursion in $\mathcal{N}$

1. Informally, a primitive recursion of $h : D \to A$ runs as follows. Assume that D has a constructor $\mathbf{c_i}$ with argument list $d_1, \ldots, d_n, \ldots$. We first apply h to each $d_1, \ldots, d_n, \ldots$, obtaining $h(d_1), \ldots, h(d_n), \ldots : A$, then we define

$$\mathbf{h(c_i(d_1, \ldots, d_n, \ldots)) = r_i(h(d_1), \ldots, h(d_n), \ldots)}$$

   The constructor $\mathbf{c_i}$ disappear.

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

3. Thus, instead of forming $h(d_1), \ldots, h(d_n), \ldots : A$, we form $h \circ f : D_i \to A$, an index map for $h(d_1) = h(f(e_1)), \ldots$, $h(d_n) = h(f(e_n)), \ldots : A$. Then we define in $\mathcal{N}$:

$$\mathbf{h(c_i(f)) = r_i(h \circ f)}$$

Definition of Extendable Recursion

1. Extendable 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 possible extensions of the data type D.

Definition of Extendable Recursion

1. Extendable recursion on D in system $\mathcal{N}$ defines a map $h : D \to A$ using one clause r_i for each D_i, and a **single extra clause** $r_n : A \to A$, dealing with all possible extensions of the data type D.
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 $\mathbf{t} \equiv \mathbf{future}(\mathbf{f}) : \mathbf{D}$ built by some new constructor **future**. We assume that $h : D \to A$ 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 $\mathbf{future}(h \circ f) : A$, 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.

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 $E \in$ **Data** to a data type $D = (D_0, \dots, D_{n-1})$: $D@E = (D_0, \dots, D_{n-1}, E) \in$ **Data**.
2. $D@E$ has one tree constructor $\mathbf{c_n} : (E \to D@E) \to D@E$ 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 $E \in$ **Data** to a data type $D = (D_0, \ldots, D_{n-1})$: $D@E = (D_0, \ldots, D_{n-1}, E) \in$ **Data**.
2. $D@E$ has one tree constructor $\mathbf{c_n} : (E \to D@E) \to D@E$ more than D.
3. $(.)@D$ is extended pointwise and componentwise to all types in **Tp** by:
   - $(A \times B)@E \equiv A@E \times B@E \in$ **Tp**
   - $(A \to B)@E \equiv A \to B@E \in$ **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 $\mathbf{future}_{m,E,D} : (E \to D) \to D$ we call **future constructors**.

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

Future constructors

In order to extend the domain of a recursive map at run-time, our **second step** is to introduce constants **future**$_{m,E,D} : (E \to D) \to D$ we call **future constructors**.

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

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

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

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

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

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 $\neg(\textbf{future}(f)) = \textbf{future}(\neg \circ f)$.

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.

$$x \equiv \mathbf{future}(\mathtt{id})$$

$0 : \mathbf{Bool}$ $\qquad$ $1 : \mathbf{Bool}$

$\mathtt{id}(0) \equiv 0$ $\qquad$ $\mathtt{id}(1) \equiv 1$

$$\rightsquigarrow \neg(x) \equiv \mathbf{future}(\neg \circ \mathtt{id})$$

$0 : \mathbf{Bool}$ $\qquad$ $1 : \mathbf{Bool}$

$\neg(\mathtt{id}(0)) \equiv 1$ $\qquad$ $\neg(\mathtt{id}(1)) \equiv 0$

Road Map: §3

- §**1.** Types of $\mathcal{N}$
- §**2.** Recursion in $\mathcal{N}$
- §**3.** Terms of $\mathcal{N}$ $\Lleftarrow$
- §**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 $\Gamma \equiv E_0, \ldots, E_{m-1}$ of data types, denoting *possible extensions* of A.
2. Terms of $\mathcal{N}$ include all algebraic combinators, pairing, projections, tree constructors, future constructors **future** : $(E \to A) \to A$ for $E \in \Gamma$, uniform application **u** : $D, (D \to D) \to D$ and for each $D = (D_0, \ldots, D_{n-1})$ a constant $\mathbf{r} \equiv \mathbf{r}_{D,A}$ denoting recursion on trees of D with result in A.
3. **r** has one recursive clause $r_i : (D_i \to A) \to A$ for each index set D_i, and one extra clause $r_n : A \to A$, dealing with extensions of A.
4. Terms are closed under application, and under the unary operator $\mathbf{Forth}_{m,E}$, 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 $n, m \in$ **Nat**, $i < n, j < m$, $D = (D_0, \ldots, D_{n-1})$, $E \in$ **Data**, $A, B \in$ **Tp**, and $\Gamma = E_0, \ldots, E_{m-1}$ any context :

1. $\Gamma \vdash C : A$. If $C(\vec{x}) = \alpha[\vec{x}]$ is a combinator of type A
2. $\Gamma \vdash < \_, \_ >: A, B \to A_1 \times A_2$ and $\Gamma \vdash \pi_i : A_1 \times A_2 \to A_i$
3. (constructors) $\Gamma \vdash \mathbf{cons}_{i,D} : (D_i \to D) \to D$
4. If $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : A$, then $\Gamma \vdash t(u) : B$.

Definition (Terms of $\mathcal{N}$)

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

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

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

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

Road Map: §4

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

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 $\mathbf{future}_E$ of index set E.
2. $\mathbf{future}_E$ has type $: (E \to D) \to D$, and has domain all maps $E \to D$.
3. If $E = D$, defining these maps requires to define D **before completing the definition** of any $t : D$.
4. Thus, the definition of $\mathbf{future}_E$ is not stratified.

Candidates and Approximated Constructors

Let $E \in$ **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 $\mathtt{j}_X : (X \to D) \to D$.
3. We call $\mathtt{j}_X$ an **approximation** of the future constructor **future**$_E : (E \to D) \to D$.

Candidates and Approximated Constructors

Let $E \in$ **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 $\mathtt{j}_X : (X \to D) \to D$.
3. We call $\mathtt{j}_X$ an **approximation** of the future constructor **future**$_E : (E \to D) \to D$.
4. The branching of $\mathtt{j}_X(f)$ is a **restriction** of the branching of **future**$_E(f)$.
5. The definition of $\mathtt{j}_X$ is stratified, therefore if we may interpret **future**$_E = \mathtt{j}_X$ we would be done.

Unfortunately . . . *(see next slide)*

A second vicious cycle

1. Unfortunately, we cannot have $\textbf{future}_E = \mathtt{j}_X$ in $\mathcal{A}$.
2. Indeed, if we choose $X \subseteq E_{\mathcal{A}}$ and we add the new constant $\mathtt{j}_X$ 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 $e \not\in X$, hence $X \neq E$ and $\textbf{future}_E \neq \mathtt{j}_X$.

A second vicious cycle

1. Unfortunately, we cannot have **future**$_E = \mathtt{j}_X$ in $\mathcal{A}$.
2. Indeed, if we choose $X \subseteq E_{\mathcal{A}}$ and we add the new constant $\mathtt{j}_X$ 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**$_E \neq \mathtt{j}_X$.
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 $\mathtt{j}_{E_\mathcal{A}}$.
2. $\mathcal{N}$ cannot distinguish between **future**$_E$ and $\mathtt{j}_{E_\mathcal{A}}$: thus, the behavior of **future**$_E$ in $\mathcal{A}$ may be described from the behavior of $\mathtt{j}_{E_\mathcal{A}}$ 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 $\mathtt{j}_{E_\mathcal{A}}$.
2. $\mathcal{N}$ cannot distinguish between **future**$_E$ and $\mathtt{j}_{E_\mathcal{A}}$: thus, the behavior of **future**$_E$ in $\mathcal{A}$ may be described from the behavior of $\mathtt{j}_{E_\mathcal{A}}$ in $\mathcal{A}^E$, without any vicious cycle.
3. By exploiting this idea we may adapt Tait's notion of reducibility ([3]) 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.

Road Map: §5

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

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 $t : D \in$* **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 $\mathtt{HA}^2$*

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. 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`

A Century of Constructive Reasoning . . .

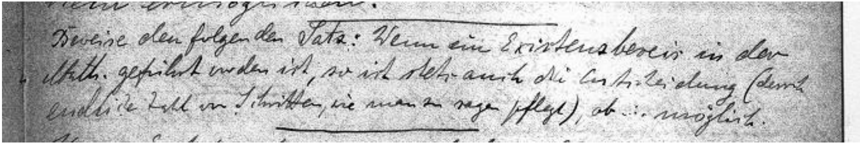

Beweise den folgenden Satz: Wenn ein Existenzbeweis in der Math. geführt worden ist, so ist stets auch die Entscheidung (durch endliche Zahl von Schritten, wie man zu sagen pflegt), ob ... möglich.

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 complete set of reductions for $\mathcal{N}$

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. $\pi_i(< a_1, a_2 >) \rightsquigarrow a_i$ 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 $c \equiv \textbf{future}_{i,E}$ 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 **u**, **r** for $\mathcal{N}$)

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

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

1. **(up-grading)** **Forth.future**$_{i,E,D}(f) \rightsquigarrow$ $\mathbf{c_{n,D@E}}$(**Forth**.f)
2. **Forth.future**$_{j+1,E,D}(f) \rightsquigarrow$ **future**$_{j,E,D@E}$(**Forth**.f) for $j \geq i$
3. **Forth**.$c(f) \rightsquigarrow c$(**Forth**.f) for any other (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 $t^{i,E}$

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. $\mathbf{Forth}_{j,F}(u)^{i,E} \equiv \mathbf{Forth}_{j,F}(u^{i+1,E})$ in all other cases.
5. $t(u)^{i,E} \equiv t^{i,E}(u^{i,E})$