A typed λ-calculus with call-by-name and call-by-value iteration

Herman Geuvers

Radboud University Nijmegen and Eindhoven University of Technology
The Netherlands

Types 2015, Tallinn
May 21, 2015

Contents

- Church data and Scott data
- Iteration, case and primitive recursion
- Call-by-value and Call-by name iteration
- Examples: addition and storage operator
- Properties (normalization) and connections

Church numerals

The most well-known Church data type

$$\begin{array}{lcl} \overline{0} & := & \lambda x\, f.x \\ \overline{1} & := & \lambda x\, f.f\, x \\ \overline{2} & := & \lambda x\, f.f\, (f\, x) \end{array} \qquad \begin{array}{lcl} \overline{p} & := & \lambda x\, f.f^{p}\, (x) \\ \mathrm{Succ} & := & \lambda n.\lambda x\, f.f\, (n\, x\, f) \end{array}$$

- The Church data types have iteration as basis. The numerals are iterators.
- Iteration scheme for Nat. (Let D be any type.)

$$\frac{d : D \quad f : D \to D}{\mathrm{It}\, d\, f : \mathrm{Nat} \to D} \quad \text{with} \quad \begin{array}{lcl} \mathrm{It}\, d\, f\, \overline{0} & \twoheadrightarrow & d \\ \mathrm{It}\, d\, f\, (\mathrm{Succ}\, x) & \twoheadrightarrow & f\, (\mathrm{It}\, d\, f\, x) \end{array}$$

- Advantage: quite a bit of well-founded recursion for free.
- Disadvantage: no pattern matching built in; predecessor is hard to define; the reduction is actually a conversion

Scott numerals

$$\begin{array}{rcl} \langle 0\rangle & := & \lambda x\, f.x \\ \langle 1\rangle & := & \lambda x\, f.f\, \langle 0\rangle \\ \langle 2\rangle & := & \lambda x\, f.f\, \langle 1\rangle \end{array} \qquad \begin{array}{rcl} \langle n+1\rangle & := & \lambda x\, f.f\, \langle n\rangle \\ \mathrm{succ} & := & \lambda p.\lambda x\, f.f\, p \end{array}$$

- The Scott numerals are case distinctors.
- Case scheme for nat. (Let D be any type.)

$$\frac{d : D \quad f : \mathrm{nat} \to D}{\mathrm{Case}\, d\, f : \mathrm{nat} \to D} \quad \text{with} \quad \begin{array}{lcl} \mathrm{Case}\, d\, f\, \langle 0\rangle & \twoheadrightarrow & d \\ \mathrm{Case}\, d\, f\, (\mathrm{succ}\, x) & \twoheadrightarrow & f\, x \end{array}$$

- $\mathrm{Case}\, d\, f := \lambda x : \mathrm{nat}.x\, d\, f$.
- Advantage: the predecessor can immediately be defined: $\mathrm{pred} := \lambda p.p\, \langle 0\rangle\, (\lambda y.y)$.
- Disadvantage: No recursion (which one has to get from somewhere else, e.g. a fixed point-combinator).

Church and Scott for general data types

Given a data type with

- constructors $\mathbf{c}_1, \ldots, \mathbf{c}_k$,
- with arity $\mathrm{ar}(i)$ of constructor $\mathbf{c}_i$.

The Church encoding is

$$\overline{\mathbf{c}_i} \quad := \quad \lambda x_1 \ldots x_{\mathrm{ar}(i)}.\lambda c_1 \ldots c_k.c_i \, (x_1 \, \vec{c}) \ldots (x_{\mathrm{ar}(i)} \, \vec{c})$$

The Scott encoding is

$$\langle \mathbf{c}_i \rangle \quad := \quad \lambda x_1 \ldots x_{\mathrm{ar}(i)}.\lambda c_1 \ldots c_k.c_i \, x_1 \ldots x_{\mathrm{ar}(i)}$$

Why study Scott data types?

- No recursion built in $\Rightarrow$ more control over the definable functions. Brunel, Terui 2010; Baillot, De Benedetti, Ronchi della Rocca 2014.
- Direct access to predecessor. Destructor is constant-time.
- Combined Church-Scott data types give Primitive Recursion scheme. For Nat, these are known as Parigot numerals. (See also Stump, Fu 2014)

$$\frac{d : D \quad f : \mathrm{Nat} \to D \to D}{\mathrm{Rec}\, d\, f : \mathrm{Nat} \to D}$$

$$\begin{array}{lcl} \mathrm{Rec}\, d\, f\, 0 & \twoheadrightarrow & d \\ \mathrm{Rec}\, d\, f\, (\mathrm{Succ}\, x) & \twoheadrightarrow & f\, x\, (\mathrm{Rec}\, d\, f\, x) \end{array}$$

One can define Rec in terms of It, but that's painful. (This is what Kleene found out at the dentist.)

Typing Church and Scott data types

- Church data types can be typed in polymorphic λ-calculus, $\lambda 2$. E.g. for Church numbers: $\mathrm{Nat} := \forall X.X \to (X \to X) \to X$.
- To type Scott data types we need $\lambda 2\mu$: $\lambda 2$ + positive recursive types:
  - $\mu X.\Phi$ is well-formed if X occurs positively in Φ.
  - Equality on types is the congruence generated from $\mu X.\Phi = \Phi[\mu X.\Phi/X]$.
  - Additional derivation rule:

$$\frac{\Gamma \vdash M : A \qquad A = B}{\Gamma \vdash M : B}$$

E.g. for Scott numerals: $\mathrm{nat} := \mu Y.\forall X.X \to (Y \to X) \to X$. We will use this as

$$\mathrm{nat} = \forall X.X \to (\mathrm{nat} \to X) \to X.$$

The system $\lambda 2\mu$It

The system $\lambda 2\mu$It combines polymorphic lambda calculus with positive recursive types and CBN and CBV iterators.

$$\mathsf{T} := \mathsf{TVar} \mid (\mathsf{T} \to \mathsf{T}) \mid \mu\mathsf{TVar}.\mathsf{T} \mid \forall\mathsf{TVar}.\mathsf{T},$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash x : B} \text{ if } A = B$$

$$\frac{\Gamma \vdash M : \forall X.A}{\Gamma \vdash M : A[B/X]} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall X.A} \text{ if } X \notin \mathrm{FV}(\Gamma)$$

The evaluation relation is weak-head reduction, $\to^{wh}$ given by

$$\frac{}{(\lambda x.M)P \;\to^{wh}\; M[P/x]} \qquad \frac{M \;\to^{wh}\; N}{M\,P \;\to^{wh}\; N\,P}$$

Plus additional typing and reduction rules for Itcbv and Itcbn.

Call-by-name and call-by-value iteration

To be able to define functions on Scott data types, we add for every data-type two program rules for iteration:

- Itcbn for call-by-name iteration from data-type D to B,
  Idea: compute first constructor of the output of type B and pass to the appropriate continuation for type B.
- Itcbv for call-by-value iteration from data-type D to B.
  Idea: compute input value completely and then pass on to a continuation of type $B \to X$.
  So, this function will be of type $\forall X.(B \to X) \to (D \to X)$
  To be abbreviated to $\forall X.\neg_X B \to \neg_X D$

Data types in $\lambda 2\mu$It

Data types have the following shape

$$D = \mu T.\forall X.(\Phi^1(T) \to X) \ldots \to (\Phi^n(T) \to X) \to X,$$

with X positive in $\Phi^i(X)$. Phrased differently:

$$D = \forall X.(\Phi^1(D) \to X) \ldots \to (\Phi^n(D) \to X) \to X,$$

Examples:

$$\begin{array}{lcl}
\mathrm{bool} & := & \forall X.X \to X \to X \\
\mathrm{nat} & := & \forall X.X \to (\mathrm{nat} \to X) \to X \\
\mathrm{list}_A & := & \forall X.X \to (A \to \mathrm{list}_A \to X) \to X \\
\mathrm{btree}_A & := & \forall X.(A \to X) \to (\mathrm{btree}_A \to \mathrm{btree}_A \to X) \to X
\end{array}$$

Constructors are defined terms in $\lambda 2\mu$It

$$\begin{array}{rcl} \mathrm{nat} & := & \forall X.X \to (\mathrm{nat} \to X) \to X \\ \mathrm{list}_A & := & \forall X.X \to (A \to \mathrm{list}_A \to X) \to X \end{array}$$

The constructors are (without type information with the λ):

$$\begin{array}{rcl} \mathrm{zero} & := & \lambda z.\lambda s.z : \mathrm{nat} \\ \mathrm{succ} & := & \lambda n.\lambda z.\lambda s.s\, n : \mathrm{nat} \to \mathrm{nat} \\ \mathrm{nil} & := & \lambda n.\lambda c.n : \mathrm{list}_A \\ \mathrm{cons} & := & \lambda a.\lambda l.\lambda n.\lambda c.c\, a\, l : A \to \mathrm{list}_A \to \mathrm{list}_A \end{array}$$

We write $\langle d \rangle$ for the encoding of a data-element as a term.
So, for $n \in \mathbf{N}$, $\langle n \rangle : \mathrm{nat}$, given by

$$\begin{array}{rcl} \langle 0 \rangle & := & \mathrm{zero} \\ \langle n+1 \rangle & := & \mathrm{succ}\, \langle n \rangle \end{array}$$

Derivation rule for Itcbn, Call-by-Name iterator

Given $D = \forall X.(\Phi^1(X) \to D) \ldots \to (\Phi^n(X) \to D) \to X$.

$$\frac{f_1 : \Phi^1(B) \to B \quad \ldots \quad f_n : \Phi^n(B) \to B}{\mathsf{Itcbn}_B^D\, f_1 \ldots f_n : D \to B}$$

Plus reduction rule

If B is also a data-type, we have
$B = \forall X.(\Psi^1(X) \to B) \ldots \to (\Psi^m(X) \to B) \to X$.
We could also write the rule for Itcbn as follows (for A an arbitrary type):

$$\frac{\begin{array}{c} f_1 : \Phi^1(B) \to B \quad \ldots \quad f_n : \Phi^n(B) \to B \quad d : D \\ c_1 : \Psi^1(B) \to A \quad \ldots \quad c_m : \Psi^m(B) \to A \end{array}}{\mathsf{Itcbn}_B^D\, f_1 \ldots f_n\, d\, c_1 \ldots c_m : A}$$

Idea: $c_1 : \Psi^1(m) \to A \ldots c_m : \Psi^m(B) \to A$ are the continuations for data type B, one for each constructor of B.

Derivation rule for Itcbv, Call-by-Value iterator

Idea: $\neg_A B := B \to A$ is the type of the continuation; $c : \neg_A B$ is called "after the value has been computed".
Let $D = \forall X.(\Phi^1(X) \to D) \ldots \to (\Phi^n(X) \to D) \to X$.

$$\frac{f_1 : \neg_A B \to \neg_A \Phi^1(B) \quad \ldots \quad f_n : \neg_A B \to \neg_A \Phi^n(B)}{\mathsf{Itcbv}_B^D\, f_1 \ldots f_n : \neg_A B \to \neg_A D}$$

Plus reduction rule

This rule can also be phrased in "fully applied form":

$$\frac{f_1 : \neg_A B \to \neg_A \Phi^1(B) \quad \ldots \quad f_n : \neg_A B \to \neg_A \Phi^n(B) \quad c : \neg_A B \quad d : D}{\mathsf{Itcbv}_B^D\, f_1 \ldots f_n\, c\, d : A}$$

Call-by-name and call-by-value iteration for nat

We show the case for $\mathrm{nat} = \forall X.X \to (\mathrm{nat} \to X) \to X$.

Call-by-name:

$$\frac{f_1 : B \quad f_2 : B \to B}{\mathsf{Itcbn}\, f_1\, f_2 : \mathrm{nat} \to B}$$

with reduction rule:

$$\mathsf{Itcbn}\, f_1\, f_2\, n \;\to\; n\, f_1\, (\lambda x : \mathrm{nat}.f_2\, (\mathsf{Itcbn}\, f_1\, f_2\, x))$$

Call-by-value:

$$\frac{f_1 : \neg_A \neg_A B \quad f_2 : \neg_A B \to \neg_A B}{\mathsf{Itcbv}\, f_1\, f_2 : \neg_A B \to \neg_A \mathrm{nat}}$$

with reduction rule:

$$\mathsf{Itcbv}\, f_1\, f_2\, c\, n \;\to\; n\, (f_1\, c)\, (\mathsf{Itcbv}\, f_1\, f_2\, (f_2\, c)).$$

Note: $f_1\, c : A$ and $\mathsf{Itcbv}\, f_1\, f_2\, (f_2\, c) : \mathrm{nat} \to A$.

Example for the case nat: Call-by-Value Addition

$$\frac{f_1 : \neg_A\neg_A\mathrm{nat} \quad f_2 : \neg_A\mathrm{nat} \to \neg_A\mathrm{nat}}{\mathsf{Itcbv}\, f_1\, f_2 : \neg_A\mathrm{nat} \to \neg_A\mathrm{nat}}$$

Define:

$\widehat{\mathrm{succ}} := \lambda c : \neg_A\mathrm{nat}.\lambda n : \mathrm{nat}.c\,(\mathrm{succ}\, n) : \neg_A\mathrm{nat} \to \neg_A\mathrm{nat}$

$\hat{n} := \lambda c : \neg_A\mathrm{nat}.c\, n : \neg_A\neg_A\mathrm{nat}$ (for $n : \mathrm{nat}$)

Then

$$\begin{array}{lcl} \mathrm{AddCBV} & : & \mathrm{nat} \to \mathrm{nat} \to \neg_A\neg_A\mathrm{nat} \\ \mathrm{AddCBV} & := & \lambda x\, y : \mathrm{nat}.\lambda c : \neg_A\mathrm{nat}.\mathsf{Itcbv}\, \hat{y}\, \widehat{\mathrm{succ}}\, c\, x \end{array}$$

We have

$$\mathrm{AddCBV}\, \langle n \rangle\, \langle m \rangle\, c \;\twoheadrightarrow\; c\, \langle n + m \rangle$$

where $\langle n \rangle$ represents the number n.

Example for the case nat: Call-by-Name Addition

$$\frac{f_1 : \mathrm{nat} \quad f_2 : \mathrm{nat} \to \mathrm{nat}}{\mathsf{Itcbn}\, f_1\, f_2 : \mathrm{nat} \to \mathrm{nat}}$$

$$\mathrm{AddCBN} := \lambda x\, y : \mathrm{nat}.\mathsf{Itcbn}\, y\, \mathrm{succ}.$$

Then

$$\begin{aligned} \mathrm{AddCBN}\, \langle n+1 \rangle\, \langle m \rangle &\twoheadrightarrow \mathrm{succ}(\mathrm{AddCBN}\, \langle n \rangle\, \langle m \rangle) \\ &\twoheadrightarrow \lambda z\, s.s(\mathrm{AddCBN}\, \langle n \rangle\, \langle m \rangle) \end{aligned}$$

- Weak-head reduction stops here.
- $\mathrm{AddCBN}\, t\, q$ computes the first 'pattern' of the output and passes it on to the appropriate continuation. (In this case s.)

Storage operators

The idea of a storage operator (Krivine, Nour), Stor, for the natural numbers nat is:

For every $n \in \mathbf{N}$ there is a term t_n such that,
given $f : \mathrm{nat} \to A$ and $M : \mathrm{nat}$ with $M =_{\beta vn} \langle n \rangle$,

$$\mathrm{Stor}\ M\ f \ \twoheadrightarrow\ f\ t_n.$$

To compute $f\ M$ call-by-value, we compute $\mathrm{Stor}\ M\ f$.
This will first compute M and then feed this into f.

Storage operators in $\lambda 2\mu$It

A storage operator for data type D is a term
Stor : $D \to \forall X.\neg_X \neg_X D$ satisfying, for all M with $M =_{\beta vn} \langle d \rangle$,

$$\mathsf{Stor}\, M\, f \;\twoheadrightarrow\; f\, \langle d \rangle.$$

Define $\mathsf{Stor}_{\mathrm{nat}} : \mathrm{nat} \to \forall X.\neg_X \neg_X \mathrm{nat}$ and
$\mathsf{Unstor}_{\mathrm{nat}} : (\forall X.\neg_X \neg_X \mathrm{nat}) \to \mathrm{nat}$ as follows.

$$\begin{aligned} \mathsf{Stor}_{\mathrm{nat}} &:= \lambda n.\lambda f.\mathsf{Itcbv}\, \widehat{\mathrm{zero}}\, \widehat{\mathrm{succ}}\, f\, n \\ \mathsf{Unstor}_{\mathrm{nat}} &:= \lambda f.\lambda z\, s.f\, (\lambda n.n\, z\, s) \end{aligned}$$

with $\widehat{\mathrm{zero}} := \lambda c.c\,\mathrm{zero}$, $\widehat{\mathrm{succ}} := \lambda c\, m.c(\mathrm{succ}\, m)$.

Lemma The term $\mathsf{Stor}_{\mathrm{nat}}$ is a storage operator for nat and
$\mathsf{Unstor}_{\mathrm{nat}}(\mathsf{Stor}_{\mathrm{nat}}\, \langle n \rangle) =_{\beta vn} \langle n \rangle$ for all $n \in \mathbf{N}$.

Strong Normalization

The proof uses the well-known saturated sets construction to give an interpretation for types as sets of (untyped) terms.

X is a saturated set ($X \in \mathrm{SAT}$) if

1. $X \subseteq \mathrm{SN}$,
2. for x a variable and $\vec{P}$ a sequence of terms, $x\vec{P} \in X$, if $x\vec{P} \in \mathrm{SN}$,
3. X is closed under reduction: if $M \in X$ and $M \twoheadrightarrow N$, then $N \in X$,
4. X is closed under weak-head-redex expansion: if $M \in X$ and $N \rightarrow^{wh} M$, then $N \in X$,

Question: how to intepret a data type as a saturated set?

- nat consists of specific closed λ-terms: zero, succ, ...
- ltcbv and ltcbn impose closure conditions on $[\![\mathrm{nat}]\!]$, and possibly on the interpretation of other types.

SAT is a complete lattice, so we will define $[\![\mathrm{nat}]\!] := \mathrm{NAT}$, where NAT is the least fixed point of some monotone $\mathcal{N} : \mathrm{SAT} \to \mathrm{SAT}$.

Intepreting nat

$[\![\mathrm{nat}]\!] := \mathsf{NAT}$, where $\mathsf{NAT} := \mathrm{lfp}(\mathcal{N})$, with $\mathcal{N} : \mathsf{SAT} \to \mathsf{SAT}$ defined by

$$\mathcal{N}(X) := \{M \in \mathrm{Term} \quad | \quad M \in \bigcap_{T \in \mathsf{SAT}} T \to (X \to T) \to T \\ \wedge \quad M \text{ satisfies (I) and (II)}\},$$

where (I) and (II) are defined by

(I) $\forall B, A \in \mathsf{SAT}\ \forall f_1 \in \neg_A \neg_A B\ \forall f_2 \in \neg_A B \to \neg_A B\ \forall c \in \neg_A B,$
$\mathsf{Itcbv}\ f_1\ f_2\ c\ M \in A$

(II) $\forall B \in \mathsf{SAT}\, \forall f_1 \in B\ \forall f_2 \in B \to B, \mathsf{Itcbn}\ f_1\ f_2\ M \in B$

Properties of $[\mathrm{nat}] := \mathsf{NAT}$

- $\mathcal{N}$ is monotone and $X \in \mathsf{SAT} \Rightarrow \mathcal{N}(X) \in \mathsf{SAT}$.
- NAT satisfies the desired fixed point equation:

$$\mathsf{NAT} = \bigcap_{T \in \mathsf{SAT}} T \to (\mathsf{NAT} \to T) \to T.$$

- As a consequence $\vdash t : \mathrm{nat} \Rightarrow t \in \mathsf{NAT}$.

- The SN proof now follows from the soundness of the SAT-model.
- The proof can be generalised to other data types.

Are Itcbv and Itcbn needed as primitives?

Equationally, Call-by-value can be defined by Call-by-name.

For the nat case assume CBN:

$$\frac{f_1 : B \quad f_2 : B \to B}{\mathsf{Itcbn}\, f_1\, f_2 : \mathrm{nat} \to B}$$

with reduction rule:

$$\mathsf{Itcbn}\, f_1\, f_2\, n \;\to\; n\, f_1\, (\lambda x : \mathrm{nat}.f_2\, (\mathsf{Itcbn}\, f_1\, f_2\, x))$$

Then

$$\frac{f_1 : \neg_A\neg_A B \qquad \dfrac{f_2 : \neg_A B \to \neg_A B}{\hat{f}_2 : \neg_A\neg_A B \to \neg_A\neg_A B}}{\dfrac{\mathsf{Itcbn}\, f_1\, \hat{f}_2 : \mathrm{nat} \to \neg_A\neg_A B}{\lambda c.\lambda n.\mathsf{Itcbn}\, f_1\, \hat{f}_2\, n\, c : \neg_A B \to \neg_A \mathrm{nat}}}$$

Define this term as $\mathsf{Itcbv}\, f_1\, f_2$. This function computes CBV and:

$$\begin{array}{lll} \mathsf{Itcbv}\, f_1\, f_2\, c\, \mathrm{zero} & \twoheadrightarrow & f_1\, c \\ \mathsf{Itcbv}\, f_1\, f_2\, c\, (\mathrm{succ}\, y) & =_{\beta vn} & \mathsf{Itcbv}\, f_1\, f_2\, (f_2\, c)\, y. \end{array}$$

Can we mediate between Scott nat and Church Nat?

- From Church to Scott: $F : \mathrm{Nat} \to \mathrm{nat}$
  $F := \lambda n.n\,\mathrm{zero}\,\mathrm{succ}$
- From Scott to Church: $G : \mathrm{nat} \to \mathrm{Nat}$
  $G :=$ Itcbn $\mathrm{Zero}\,\mathrm{Succ}$
- From Scott to Church, call-by-value: $H : \neg_A\mathrm{Nat} \to \neg_A\mathrm{nat}$
  $H :=$ Itcbv $\widehat{\mathrm{Zero}}\,\widehat{\mathrm{Succ}}$
  with $\widehat{\mathrm{Zero}} : \neg_A\neg_A\mathrm{Nat}$, $\widehat{\mathrm{Succ}} : \neg_A\mathrm{Nat} \to \neg_A\mathrm{Nat}$.

With F, we can transform any function on Scott numerals into a (call-by-name) function on Church numerals:

$$\text{If } f : \mathrm{nat} \to A \text{ then } f \circ F : \mathrm{Nat} \to A$$

For encoded numerals we have

$$f(F\overline{n}) =_{\beta vn} f\langle n\rangle$$

So all function representable on nat are representable on Nat.

Can we mediate between Scott nat and Church Nat?

- From Scott to Church: $G : \mathrm{nat} \to \mathrm{Nat}$
  $G := \mathsf{Itcbn}\,\mathrm{Zero}\,\mathrm{Succ}$
- From Scott to Church, call-by-value: $H : \neg_A\mathrm{Nat} \to \neg_A\mathrm{nat}$
  $H := \mathsf{Itcbv}\,\widehat{\mathrm{Zero}}\,\widehat{\mathrm{Succ}}$
  with $\widehat{\mathrm{Zero}} : \neg_A\neg_A\mathrm{Nat}$, $\widehat{\mathrm{Succ}} : \neg_A\mathrm{Nat} \to \neg_A\mathrm{Nat}$.

With H, we can transform any function on Church numerals into a (call-by-value) function on Scott numerals:

$$\text{If } f : \mathrm{Nat} \to A \text{ then } H\,f : \mathrm{nat} \to A$$

With G, we can transform any function on Church numerals into a (call-by-name) function on Scott numerals:

$$\text{If } f : \mathrm{Nat} \to A \text{ then } f \circ G : \mathrm{nat} \to A$$

For encoded numerals we have

$$f(G\langle n\rangle) =_{\beta vn} f\overline{n}$$

So all functions representable on Nat are representable on nat.

Can we define a constant-time predecessor for Church Nat?

Remember that $\mathrm{pred} : \mathrm{nat} \to \mathrm{nat}$ is defined by

$$\mathrm{pred}\, n := n\, \mathrm{zero}\, (\lambda x.x).$$

So we can define $\mathrm{Pred} : \mathrm{Nat} \to \mathrm{Nat}$ by composing

$$\mathrm{Nat} \xrightarrow{F} \mathrm{nat} \xrightarrow{\mathrm{pred}} \mathrm{nat} \xrightarrow{G} \mathrm{Nat}$$

- NOT in general $\mathrm{Pred}(\mathrm{Succ}\, t) =_{\beta vn} t$.
- $\mathrm{Pred}\overline{n+1} \twoheadrightarrow \overline{n}$ (for encodings of numbers).
- $\mathrm{Pred}\overline{n+1} \twoheadrightarrow \overline{n}$ in $O(n)$ steps.

Open question: can we improve on this? (Or can we prove it cannot be improved?)

Conclusion and future work

- Given a fine analysis of CBV and CBN on data types using the Scott data type definition: proper reduction behaviour, confluent, normalizing, typed storage operator.
- Same representable functions, but (?) better algorithmic behaviour?
- Flexibly combining CBN and CBV functions
- Can we define the constant time predecessor on Church Nat in $\lambda 2\mu$It?
- Can we be more refined, by using linear types and taking complexity into account? (cf. Brunel, Terui, many others).