(Typed) λ-Calculus and Formalising Mathematics à la de Bruijn

In honour of De Bruijn's 90th anniversary

Fairouz Kamareddine (Heriot-Watt University)

5 September 2008

De Bruijn in 1967

- In 1967, an internationally renowned mathematician called N.G. de Bruijn wanted to do something never done before: use the computer to formally check the correctness of mathematical books.
- Such a task needs a good formalisation of mathematics, a good competence in implementation, and extreme attention to all the details so that nothing is left informal.
- No earlier formalisation of mathematics (Frege, Russell, Whitehead, Hilbert, Ramsey, etc.) had ever achieved such attention to details.
- Implementing extensive formal systems on the computer was never done before.
- De Bruijn, an extremely original mathematician, did every step his own way, quickly replacing existing ideas with his own original genious way of thinking, shaping the road ahead for everyone else to follow.

De Bruijn in 1967

- When de Bruijn announced his new project Automath at the start of January 1967, there was mixed reactions:
  - We are talking about respected N.G. de Bruijn who is known to produce ideas of gold so the project was bound to gain support.
  - The project is so new, so revolutionary, so, it obviously gained opposition:
    - Amongst mathematicians: Why is de Bruijn defecting?
    - Amongst computer scientists: De Bruijn is not a computer scientist so why is he coming to do a computer scientist's job?
    - Amongst logicians: De Bruijn is not a logician and has he also forgotten about Goedel's undecidability results?
- But, de Bruijn was ahead of everyone else. He is a living genious whose even scribbled ideas require a life devotion from us ordinary mortal researchers.

- It goes without saying that de Bruijn and his Automath shaped the way.
- It is known that de Bruijn is the father and initiator of Logical frameworks.
- In particular, de Bruijn's Automath was hugely influential in the development of the Edinburgh Logical Frameworks.
- Constable's Nuprl project has been throughout its development, closely connected to ideas in de Bruijn's Automath (e.g., telescopes).
- Coquand and Huet's calculus of constructions and consequently the proof checker Coq were heavily influenced by de Bruijn's dependent types, PAT and Automath.
- De Bruijn was the first to put the Propositions As Types (PAT) idea in practice.
- Barendregt's cube and Pure Type systems are a beautiful example of the generality of the typing rules of Automath.

- De Bruijn was the first to express the importance of definitions to the formalisation and proof checking of mathematics. Definitions (also known as let expressions) have been adopted in other proof checkers and in programming languges (e.g. ML).
- De Bruijn's Automath was the first (and remains the only) proof checker in which an entire book has been fully proof checked by the computer (Mizar is the next system in which 60% of a book is proof checked).
- There are numerous other gems in de Bruijn's Automath.
- It has been, and will be for many generations to come, a hard but magical task to fully decode the genious ideas of de Bruijn in his Automath project.
- In this talk, I will concentrate on some low level details which explain why every sentence in de Bruijn's writing is a gem that can help us solve numerous problems in current research in programming language theory/implementation and in logical frameworks.

De Bruijn Indices [de Bruijn, 1972]

- Classical λ-calculus: $A ::= x \mid (\lambda x.B) \mid (BC)$
  $(\lambda x.A)B \to_\beta A[x := B]$
- $(\lambda x.\lambda y.xy)y \to_\beta (\lambda y.xy)[x := y] \neq \lambda y.yy$
- $(\lambda x.\lambda y.xy)y \to_\beta (\lambda y.xy)[x := y] =_\alpha (\lambda z.xz)[x := y] = \lambda z.yz$
- $\lambda x.x$ and $\lambda y.y$ are the same function. Write this function as $\lambda 1$.
- Assume a free variable list (say $x, y, z, \dots$).
- $(\lambda\lambda 2\, 1)2 \to_\beta (\lambda 2\, 1)[1 := 2] = \lambda(2[2 := 3])(1[2 := 3]) = \lambda 3\, 1$

Classical λ-calculus with de Bruijn indices

- Let $i, n \geq 1$ and $k \geq 0$
- $A ::= n \mid (\lambda B) \mid (BC)$
  $(\lambda A)B \to_\beta A\{\!\{1 \leftarrow B\}\!\}$
- $$U_k^i(AB) = U_k^i(A)\, U_k^i(B) \qquad U_k^i(\lambda A) = \lambda(U_{k+1}^i(A))$$
  $$U_k^i(\mathtt{n}) = \begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k\,. \end{cases}$$
- $$\begin{aligned} (A_1A_2)\{\!\{\mathtt{i} \leftarrow B\}\!\} &= (A_1\{\!\{\mathtt{i} \leftarrow B\}\!\})\,(A_2\{\!\{\mathtt{i} \leftarrow B\}\!\}) \\ (\lambda A)\{\!\{\mathtt{i} \leftarrow B\}\!\} &= \lambda(A\{\!\{\mathtt{i+1} \leftarrow B\}\!\}) \end{aligned}$$
  $$\mathtt{n}\{\!\{\mathtt{i} \leftarrow B\}\!\} = \begin{cases} \mathtt{n-1} & \text{if } n > i \\ U_0^i(B) & \text{if } n = i \\ \mathtt{n} & \text{if } n < i\,. \end{cases}$$
- Numerous implementations of proof checkers and programming languages have been based on de Bruijn indices.

From classical λ-calculus with de Bruijn indices to substitution calculus λs [Kamareddine and Ríos, 1995]

- Write $A\{\!\{n \leftarrow B\}\!\}$ as $A\sigma^n B$ and $U^i_k(A)$ as $\varphi^i_k A$.
- $A ::= n \mid (\lambda B) \mid (BC) \mid (A\sigma^i B) \mid (\varphi^i_k B) \quad where \quad i, n \geq 1\,, \quad k \geq 0\,.$

σ-generation	$(\lambda A)\, B$	$\longrightarrow$	$A\,\sigma^1\, B$
σ-λ-transition	$(\lambda A)\, \sigma^i B$	$\longrightarrow$	$\lambda(A\,\sigma^{i+1}\, B)$
σ-app-transition	$(A_1\, A_2)\, \sigma^i B$	$\longrightarrow$	$(A_1\, \sigma^i B)\, (A_2\, \sigma^i B)$
σ-destruction	$\mathtt{n}\, \sigma^i B$	$\longrightarrow$	$\begin{cases} \mathtt{n-1} & \text{if } n > i \\ \varphi^i_0\, B & \text{if } n = i \\ \mathtt{n} & \text{if } n < i \end{cases}$
φ-λ-transition	$\varphi^i_k(\lambda A)$	$\longrightarrow$	$\lambda(\varphi^i_{k+1}\, A)$
φ-app-transition	$\varphi^i_k(A_1\, A_2)$	$\longrightarrow$	$(\varphi^i_k\, A_1)\, (\varphi^i_k\, A_2)$
φ-destruction	$\varphi^i_k\, \mathtt{n}$	$\longrightarrow$	$\begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k \end{cases}$

1. The s-calculus (i.e., λs minus σ-generation) is strongly normalising,
2. The λs-calculus is confluent and simulates (in small steps) β-reduction
3. The λs-calculus preserves strong normalisation PSN.
4. The λs-calculus has a confluent extension with open terms λse.

- The λs-calculus is the only calculus of substitutions which satisfies all the above properties 1., 2., 3. and 4.

$\lambda\upsilon$ [Benaissa et al., 1996]

Terms: $\Lambda\upsilon^t ::= \mathbb{N} \mid \Lambda\upsilon^t\Lambda\upsilon^t \mid \lambda\Lambda\upsilon^t \mid \Lambda\upsilon^t[\Lambda\upsilon^s]$
Substitutions: $\Lambda\upsilon^s ::= \uparrow \mid\ \Uparrow(\Lambda\upsilon^s) \mid \Lambda\upsilon^t$.

(Beta)	$(\lambda a)\, b$	$\longrightarrow$	$a\,[b/]$
(App)	$(a\,b)[s]$	$\longrightarrow$	$(a\,[s])\,(b\,[s])$
(Abs)	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a\,[\Uparrow(s)])$
(FVar)	$\mathtt{1}\,[a/]$	$\longrightarrow$	a
(RVar)	$\mathtt{n}+\mathtt{1}\,[a/]$	$\longrightarrow$	$\mathtt{n}$
(FVarLift)	$\mathtt{1}\,[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{1}$
(RVarLift)	$\mathtt{n}+\mathtt{1}\,[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{n}\,[s]\,[\uparrow]$
(VarShift)	$\mathtt{n}\,[\uparrow]$	$\longrightarrow$	$\mathtt{n}+\mathtt{1}$

$\lambda\upsilon$ satisfies 1., 2., and 3., but does not have a confluent extension on open terms.

$\lambda\sigma_{\Uparrow}$

Terms: $\Lambda\sigma^t_{\Uparrow} ::= \mathbb{N} \mid \Lambda\sigma^t_{\Uparrow}\Lambda\sigma^t_{\Uparrow} \mid \lambda\Lambda\sigma^t_{\Uparrow} \mid \Lambda\sigma^t_{\Uparrow}[\Lambda\sigma^s_{\Uparrow}]$

Substitutions: $\Lambda\sigma^s_{\Uparrow} ::= id \mid \uparrow \mid \Uparrow(\Lambda\sigma^s_{\Uparrow}) \mid \Lambda\sigma^t_{\Uparrow} \cdot \Lambda\sigma^s_{\Uparrow} \mid \Lambda\sigma^s_{\Uparrow} \circ \Lambda\sigma^s_{\Uparrow}$.

(Beta)	$(\lambda a)\, b$	$\longrightarrow$	$a\,[b \cdot id]$
(App)	$(a\, b)[s]$	$\longrightarrow$	$(a\,[s])\,(b\,[s])$
(Abs)	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a\,[\Uparrow(s)])$
(Clos)	$(a\,[s])[t]$	$\longrightarrow$	$a\,[s \circ t]$
(Varshift1)	$\mathtt{n}\,[\uparrow]$	$\longrightarrow$	$\mathtt{n}+1$
(Varshift2)	$\mathtt{n}\,[\uparrow \circ\, s]$	$\longrightarrow$	$\mathtt{n}+1\,[s]$
(FVarCons)	$1\,[a \cdot s]$	$\longrightarrow$	a
(RVarCons)	$\mathtt{n}+1\,[a \cdot s]$	$\longrightarrow$	$\mathtt{n}\,[s]$
(FVarLift1)	$1\,[\Uparrow(s)]$	$\longrightarrow$	1
(FVarLift2)	$1\,[\Uparrow(s) \circ t]$	$\longrightarrow$	$1\,[t]$
(RVarLift1)	$\mathtt{n}+1\,[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{n}[s \circ \uparrow]$
(RVarLift2)	$\mathtt{n}+1\,[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{n}[s \circ (\uparrow \circ\, t)]$

$\lambda\sigma_{\Uparrow}$ rules continued

(Map)	$(a \cdot s) \circ t$	$\longrightarrow$	$a[t] \cdot (s \circ t)$
(Ass)	$(s \circ t) \circ u$	$\longrightarrow$	$s \circ (t \circ u)$
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s
(ShiftLift1)	$\uparrow \circ \Uparrow(s)$	$\longrightarrow$	$s \circ \uparrow$
(ShiftLift2)	$\uparrow \circ (\Uparrow(s) \circ t)$	$\longrightarrow$	$s \circ (\uparrow \circ t)$
(Lift1)	$\Uparrow(s) \circ \Uparrow(t)$	$\longrightarrow$	$\Uparrow(s \circ t)$
(Lift2)	$\Uparrow(s) \circ (\Uparrow(t) \circ u)$	$\longrightarrow$	$\Uparrow(s \circ t) \circ u$
(LiftEnv)	$\Uparrow(s) \circ (a \cdot t)$	$\longrightarrow$	$a \cdot (s \circ t)$
(IdL)	$id \circ s$	$\longrightarrow$	s
(IdR)	$s \circ id$	$\longrightarrow$	s
(LiftId)	$\Uparrow(id)$	$\longrightarrow$	id
(Id)	$a[id]$	$\longrightarrow$	a

$\lambda\sigma_{\Uparrow}$ satisfies 1., 2., and 4., but does not have PSN.

How is λse obtained from λs?

- $A ::= X \mid n \mid (\lambda B) \mid (BC) \mid (A\sigma^i B) \mid (\varphi^i_k B) \quad where \quad i, n \geq 1\,, \;\; k \geq 0\,.$

- Extending the syntax with open terms without extending then rules loses the confluence (even local confluence):
  $((\lambda X)Y)\sigma^1 1 \to (X\sigma^1 Y)\sigma^1 1 \qquad ((\lambda X)Y)\sigma^1 1 \to ((\lambda X)\sigma^1 1)(Y\sigma^1 1)$

- $(X\sigma^1 Y)\sigma^1 1$ and $((\lambda X)\sigma^1 1)(Y\sigma^1 1)$ have no common reduct.

- But, $((\lambda X)\sigma^1 1)(Y\sigma^1 1) \twoheadrightarrow (X\sigma^2 1)\sigma^1(Y\sigma^1 1)$

- Simple: add de Bruijn's metasubstitution and distribution lemmas to the rules of λs:

σ-σ	$(A\sigma^i B)\,\sigma^j\,C$	$\longrightarrow$	$(A\,\sigma^{j+1}\,C)\,\sigma^i\,(B\,\sigma^{j-i+1}\,C)$	if	$i \leq j$
σ-φ *1*	$(\varphi^i_k\,A)\,\sigma^j\,B$	$\longrightarrow$	$\varphi^{i-1}_k\,A$	if	$k < j < k+i$
σ-φ *2*	$(\varphi^i_k\,A)\,\sigma^j\,B$	$\longrightarrow$	$\varphi^i_k(A\,\sigma^{j-i+1}\,B)$	if	$k+i \leq j$
φ-σ	$\varphi^i_k(A\,\sigma^j\,B)$	$\longrightarrow$	$(\varphi^i_{k+1}\,A)\,\sigma^j\,(\varphi^i_{k+1-j}\,B)$	if	$j \leq k+1$
φ-φ *1*	$\varphi^i_k\,(\varphi^j_l\,A)$	$\longrightarrow$	$\varphi^j_l\,(\varphi^i_{k+1-j}\,A)$	if	$l+j \leq k$
φ-φ *2*	$\varphi^i_k\,(\varphi^j_l\,A)$	$\longrightarrow$	$\varphi^{j+i-1}_l\,A$	if	$l \leq k < l+j$

- These extra rules are the rewriting of the well-known meta-substitution $(\sigma - \sigma)$ and distribution $(\varphi - \sigma)$ lemmas (and the 4 extra lemmas needed to prove them).

Where did the extra rules come from?

In de Bruijn's classical λ-calculus we have the lemmas:

$(\sigma - \varphi\ 1)$ For $k < j < k+i$ we have: $U_k^{i-1}(A) = U_k^i(A)\{\!\{ \mathtt{j} \leftarrow B \}\!\}$.

$(\varphi - \varphi\ 2)$ For $l \leq k < l+j$ we have: $U_k^i(U_l^j(A)) = U_l^{j+i-1}(A)$.

$(\sigma - \varphi\ 2)$ For $k+i \leq j$ we have: $U_k^i(A)\{\!\{ \mathtt{j} \leftarrow B \}\!\} = U_k^i(A\{\!\{ \mathtt{j} - \mathtt{i} + \mathtt{1} \leftarrow B \}\!\})$.

$(\sigma - \sigma)$ *[Meta-substitution lemma]* For $i \leq j$ we have:
$A\{\!\{ \mathtt{i} \leftarrow B \}\!\}\{\!\{ \mathtt{j} \leftarrow C \}\!\} = A\{\!\{ \mathtt{j} + \mathtt{1} \leftarrow C \}\!\}\{\!\{ \mathtt{i} \leftarrow B\{\!\{ \mathtt{j} - \mathtt{i} + \mathtt{1} \leftarrow C \}\!\} \}\!\}$.

$(\varphi - \varphi\ 1)$ For $j \leq k+1$ we have: $U_{k+p}^i(U_p^j(A)) = U_p^j(U_{k+p+1-j}^i(A))$.

$(\varphi - \sigma)$ *[Distribution lemma]*
For $j \leq k+1$ we have: $U_k^i(A\{\!\{ \mathtt{j} \leftarrow B \}\!\}) = U_{k+1}^i(A)\{\!\{ \mathtt{j} \leftarrow U_{k+1-j}^i(B) \}\!\}$.

The proof of $(\sigma - \sigma)$ uses $(\sigma - \varphi\ 1)$ and $(\sigma - \varphi\ 2)$ both with $k = 0$.
The proof of $(\sigma - \varphi\ 2)$ requires $(\varphi - \varphi\ 2)$ with $l = 0$.
Finally, $(\varphi - \varphi\ 1)$ with $p = 0$ is needed to prove $(\varphi - \sigma)$).

Summary of [de Bruijn, 1972]

- Those who in 1967 said that de Bruijn was not a computer scientist, must have felt embarassed afterwards.
- De Bruijn's work not only had serious consequences on the implementation of numerous programming languages and proof checkers, but also was to dominate decades of research and hundreds of articles by many researchers.
- Here, I described how his work in [de Bruijn, 1972] influenced research into calculi of explicit substitutions.
- Next I describe how his novel notation for the λ-calculus influenced notions of generalised reductions (with implications on theoretical and implementational aspects of the λ-calculus and programming languages).

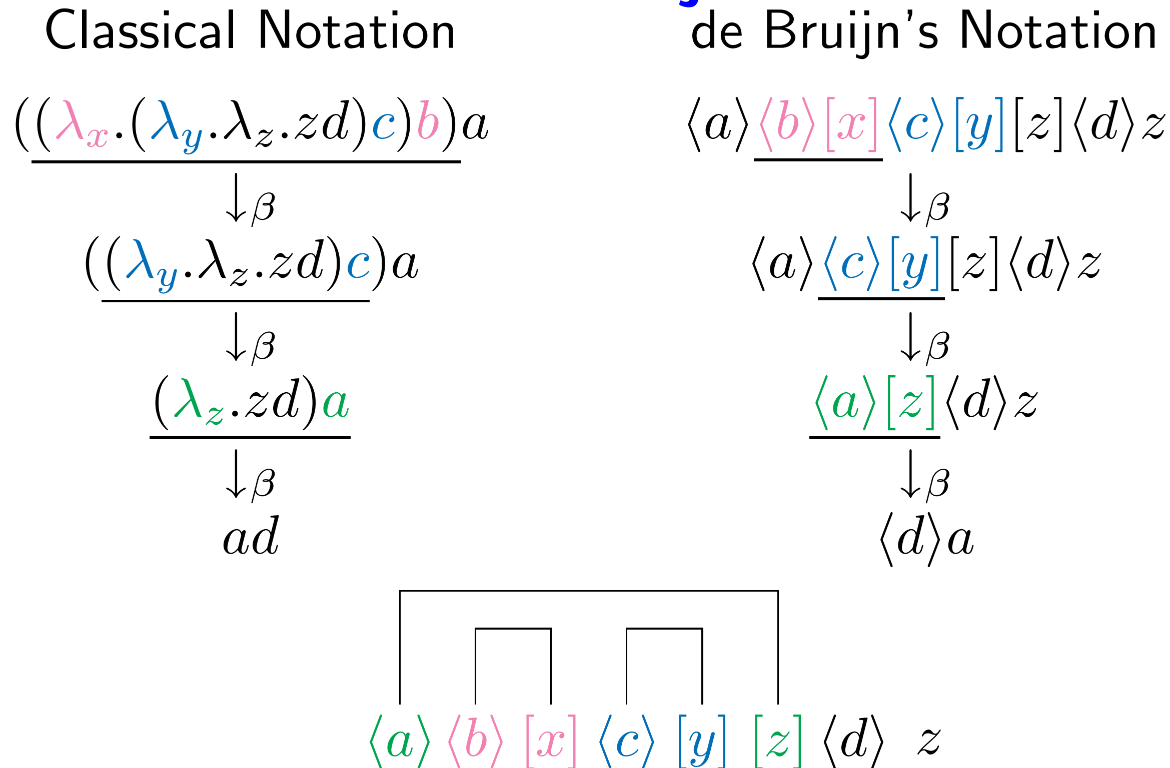
Lambda Calculus à la de Bruijn

- $\mathcal{I}(x) = x, \qquad \mathcal{I}(\lambda x.B) = [x]\mathcal{I}(B), \qquad \mathcal{I}(AB) = \langle \mathcal{I}(B) \rangle \mathcal{I}(A)$
- $(\lambda x.\lambda y.xy)z$ translates to $\langle z \rangle [x][y] \langle y \rangle x$.
- The *applicator wagon* $\langle z \rangle$ and *abstractor wagon* $[x]$ occur NEXT to each other.
- $(\lambda x.A)B \to_\beta A[x := B]$ becomes

$$\langle B \rangle [x] A \to_\beta [x := B]A$$

- The "bracketing structure" of $((\lambda_x.(\lambda_y.\lambda_z.--)c)b)a)$, is '$[_1 \ [_2 \ [_3 \ ]_2 \ ]_1 \ ]_3$', where '$[_i$' and '$]_i$' match.
- The bracketing structure of $\langle a \rangle \langle b \rangle [x] \langle c \rangle [y][z] \langle d \rangle$ is simpler: $[[\,][\,]]$.
- $\langle b \rangle [x]$ and $\langle c \rangle [y]$ are AT-pairs whereas $\langle a \rangle [z]$ is an AT-couple.

Redexes in de Bruijn's notation



De Bruijn was the first to introduce local/global/mini reductions into the λ-calculus.

For further details see: [Bruijn, 1984].

Further study of de Bruijn's notation can be found in [Kamareddine and Nederpelt, 1995, 1996]

Some notions of reduction studied in the literature

Name	In Classical Notation	In de Bruijn's notation
(θ)	$((\lambda_x.N)P)Q$ $\downarrow$ $(\lambda_x.NQ)P$	$\langle Q\rangle\langle P\rangle[x]N$ $\downarrow$ $\langle P\rangle[x]\langle Q\rangle N$
(γ)	$(\lambda_x.\lambda_y.N)P$ $\downarrow$ $\lambda_y.(\lambda_x.N)P$	$\langle P\rangle[x][y]N$ $\downarrow$ $[y]\langle P\rangle[x]N$
(γ_C)	$((\lambda_x.\lambda_y.N)P)Q$ $\downarrow$ $(\lambda_y.(\lambda_x.N)P)Q$	$\langle Q\rangle\langle P\rangle[x][y]N$ $\downarrow$ $\langle Q\rangle[y]\langle P\rangle[x]N$
(g)	$((\lambda_x.\lambda_y.N)P)Q$ $\downarrow$ $(\lambda_x.N[y:=Q])P$	$\langle Q\rangle\langle P\rangle[x][y]N$ $\downarrow$ $\langle P\rangle[x][y:=Q]N$

A Few Uses of these reductions/term reshuffling

- Regnier [1992] uses θ and γ in analyzing perpetual reduction strategies.
- Term reshuffling is used in [Kfoury et al., 1994; Kfoury and Wells, 1994] in analyzing typability problems.
- [Nederpelt, 1973; de Groote, 1993; Kfoury and Wells, 1995] use generalised reduction and/or term reshuffling in relating SN to WN.
- [Ariola et al., 1995] uses a form of term-reshuffling in obtaining a calculus that corresponds to lazy functional evaluation.
- [Kamareddine and Nederpelt, 1995; Kamareddine et al., 1999, 1998; Bloo et al., 1996] shows that they could reduce space/time needs.
- [Kamareddine, 2000] shows the conservation theorem for generalised reduction.
- All these works have been heavily influenced by de Bruijn's Automath whose λ-notation facilitated the manipulation of redexes.

Canonical Forms

- Nice canonical forms look like:

bachelor [ ]s	⟨⟩[ ]-pairs, A_i in CF	bachelor ⟨⟩s, B_i in CF	end var
$[x_1] \ldots [x_n]$	$\langle A_1\rangle[y_1] \ldots \langle A_m\rangle[y_m]$	$\langle B_1\rangle \ldots \langle B_p\rangle$	x

- classical:

$$\lambda x_1 \cdots \lambda x_n.(\lambda y_1.(\lambda y_2. \cdots (\lambda y_m.xB_p \cdots B_1)A_m \cdots)A_2)A_1$$

- For example, a canonical form of:

$$[x][y]\langle a\rangle[z][x']\langle b\rangle\langle c)\langle d\rangle[y'][z']\langle e\rangle x$$

is

$$[x][y][x']\langle a\rangle[z]\langle d\rangle[y']\langle c\rangle[z']\langle b\rangle\langle e\rangle x$$

Obtaining Canonical Forms

For $M \equiv [x][y]\langle a\rangle[z][x']\langle b\rangle\langle c\rangle\langle d\rangle[y'][z']\langle e\rangle x$:

$\theta(M)$:	bach. $[\,]$s $[x][y]$	$\langle\rangle[\,]$-pairs mixed with bach. $[\,]$s $\langle a\rangle[z][x']\langle d\rangle[y']\langle c\rangle[z']$	bach. $\langle\rangle$s $\langle b\rangle\langle e\rangle$	end var x
$\gamma(M)$:	bach. $[\,]$s $[x][y][x']$	$\langle\rangle[\,]$-pairs mixed with bach. $\langle\rangle$s $\langle a\rangle[z]\langle b\rangle\langle c\rangle[z']\langle d\rangle[y']$	bach. $\langle\rangle$s $\langle e\rangle$	end var x
$\theta(\gamma(M))$:	bach. $[\,]$s $[x][y][x']$	$\langle\rangle[\,]$-pairs $\langle a\rangle[z]\langle c\rangle[z']\langle d\rangle[y']$	bach. $\langle\rangle$s $\langle b\rangle\langle e\rangle$	end var x
$\gamma(\theta(M))$:	bach. $[\,]$s $[x][y][x']$	$\langle\rangle[\,]$-pairs $\langle a\rangle[z]\langle d\rangle[y']\langle c\rangle[z']$	bach. $\langle\rangle$s $\langle b\rangle\langle e\rangle$	end var x

Classes of terms modulo reductional behaviour

- Both $\theta(\gamma(A))$ and $\gamma(\theta(A))$ are in *canonical form*.

- $\theta(\gamma(A)) =_p \gamma(\theta(A))$ where $\rightarrow_p$ is the rule

$$\langle A_1 \rangle [y_1] \langle A_2 \rangle [y_2] B \rightarrow_p \langle A_2 \rangle [y_2] \langle A_1 \rangle [y_1] B \qquad \text{if } y_1 \notin \mathrm{FV}(A_2)$$

- We define: $[A]$ to be $\{B \mid \theta(\gamma(A)) =_p \theta(\gamma(B))\}$.

- One-step class-reduction $\leadsto_\beta$ is the least compatible relation such that:

$$A \leadsto_\beta B \quad \text{iff} \quad \exists A' \in [A]. \exists B' \in [B].\ A' \rightarrow_\beta B'$$

- $\leadsto_\beta$ *really acts as reduction on classes:*

- If $A \leadsto_\beta B$ then forall $A' \in [A]$, forall $B' \in [B]$, we have $A' \leadsto_\beta B'$.

Properties of reduction modulo classes

- *$\rightsquigarrow_\beta$ generalises $\to_g$ and $\to_\beta$:* $\to_\beta \subset \to_g \subset \rightsquigarrow_\beta \subset =_\beta$.
- *$\approx_\beta$ and $=_\beta$ are equivalent:* $A \approx_\beta B$ iff $A =_\beta B$.
- *$\rightsquigarrow\!\!\!\!\to_\beta$ is Church Rosser:*
  If $A \rightsquigarrow\!\!\!\!\to_\beta B$ and $A \rightsquigarrow\!\!\!\!\to_\beta C$, then for some D: $B \rightsquigarrow\!\!\!\!\to_\beta D$ and $C \rightsquigarrow\!\!\!\!\to_\beta D$.
- *Classes preserve $SN_{\to_\beta}$:* If $A \in SN_{\to_\beta}$ and $A' \in [A]$ then $A' \in SN_{\to_\beta}$.
- *Classes preserve $SN_{\rightsquigarrow_\beta}$:* If $A \in SN_{\rightsquigarrow_\beta}$ and $A' \in [A]$ then $A' \in SN_{\rightsquigarrow_\beta}$.
- *$SN_{\to_\beta}$ and $SN_{\rightsquigarrow_\beta}$ are equivalent:* $A \in SN_{\rightsquigarrow_\beta}$ iff $A \in SN_{\to_\beta}$.

Typed λ-calculus with single binder

- Let $f : \mathbb{N} \to \mathbb{N}$ et $g_f : \mathbb{N} \to \mathbb{N}$ such that $g_f(x) = f(f(x))$.
  Let $F_{\mathbb{N}} : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ such that $F_{\mathbb{N}}(f)(x) = g_f(x) = f(f(x))$.
- In Church's simply typed lambda calculus we write the function $F_{\mathbb{N}}$ as follows:

$$\lambda_{f:\mathbb{N}\to\mathbb{N}}.\lambda_{x:\mathbb{N}}.f(f(x))$$

- If we want the same function on the booleans $\mathcal{B}$, we write:

the function $F_{\mathcal{B}}$ is	$\lambda_{f:\mathcal{B}\to\mathcal{B}}.\lambda_{x:\mathcal{B}}.f(f(x))$
the type of the function $F_{\mathcal{B}}$ is	$(\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B})$

Polymorphism: the typed λ-calculus after Church

- Instead of repeating the work, we take: $\alpha : *$ (α is an arbitrary type) and we define a polymorphic function F as follows:

  $$\lambda_{\alpha:*}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f(x))$$

  We give F the type:

  $$\Pi_{\alpha:*}.(\alpha \to \alpha) \to (\alpha \to \alpha)$$

- This way, $F(\alpha) = \lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f(x)) : (\alpha \to \alpha) \to (\alpha \to \alpha)$

- We can instantiate α according to our need:
  - $F(\mathbb{N}) = \lambda_{f:\mathbb{N}\to\mathbb{N}}.\lambda_{x:\mathbb{N}}.f(f(x)) : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$
  - $F(\mathcal{B}) = \lambda_{f:\mathcal{B}\to\mathcal{B}}.\lambda_{x:\mathcal{B}}.f(f(x)) : (\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B})$
  - $F(\mathcal{B} \to \mathcal{B}) = \lambda_{f:(\mathcal{B}\to\mathcal{B})\to(\mathcal{B}\to\mathcal{B})}.\lambda_{x:(\mathcal{B}\to\mathcal{B})}.f(f(x)) :$
    $((\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B})) \to ((\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B}))$

- This way, types are like functions:
  - We can form them by abstraction
  - We can instantiate them

- But in the passage from simple to polymorphic types, we have forgotten to adapt the rule:

$$(\beta) \qquad (\lambda_{x:A}.b)C \rightarrow b[x := C]$$

  to a rule which resembles Π:

$$(\Pi) \qquad (\Pi_{x:A}.B)C \rightarrow B[x := C]$$

- Usually, if $b : B$, we take $(\lambda_{x:A}.b)C : B[x := C]$ instead of $(\lambda_{x:A}.b)C : (\Pi_{x:A}.B)C$

- De Bruijn's Automath shows that the rule Π is useful.

- It seems that the development of type theory is taking us more and more towards adopting a similar role for the binders λ et Π.

- Did we really need to separate λ et Π?

- I believe that the separation is artificial and that de Bruijn's intuition is again a winner.

- What are the properties of type theories with a single binder which represents both λ et Π?

Notation of Modern type systems

- Let $\mathcal{V} = \mathcal{V}^* \cup \mathcal{V}^\Box$ where $\mathcal{V}^* \cap \mathcal{V}^\Box = \emptyset$. Let $s \in \{*, \Box\}$.

  $\mathcal{T} ::= s \mid \mathcal{V} \mid \lambda_{\mathcal{V}:\mathcal{T}}.\mathcal{T} \mid \Pi_{\mathcal{V}:\mathcal{T}}.\mathcal{T} \mid \mathcal{T}\mathcal{T}$

  $\mathcal{T}_\flat ::= s \mid \mathcal{V} \mid \flat_{\mathcal{V}:\mathcal{T}_\flat}.\mathcal{T}_\flat \mid \mathcal{T}_\flat\mathcal{T}_\flat$

  Note that even though modern type systems have not identified the binders λ et Π, they have adapted (in $\mathcal{T}$) a common syntax in some sense.

- If $A \in \mathcal{T}$, we define $\overline{A} \in \mathcal{T}_\flat$ by: $\overline{s} \equiv s$, $\overline{x} \equiv x$, $\overline{AB} \equiv \overline{A}\,\overline{B}$, $\overline{\lambda_{x:A}.B} \equiv \overline{\Pi_{x:A}.B} \equiv \flat_{x:\overline{A}}.\overline{B}$.

- If $A \in \mathcal{T}_\flat$, we define $A^\lambda \in \mathcal{T}$ by: $s^\lambda \equiv s$, $x^\lambda \equiv x$, $(AB)^\lambda \equiv A^\lambda B^\lambda$, $(\flat_{x:A}.B)^\lambda \equiv \lambda_{x:A^\lambda}.B^\lambda$. We define $[A]$ by $\{A'$ in $\mathcal{T} \mid \overline{A'} \equiv A\}$.

- A context Γ is of the form: : $x_1 : A_1, \ldots, x_n : A_n$. $\mathrm{DOM}\,(\Gamma) = \{x_1, \ldots x_n\}$.

- In $\mathcal{T}$ we define: $\overline{\langle\rangle} \equiv \langle\rangle$ $\overline{\Gamma, x : A} \equiv \overline{\Gamma}, x : \overline{A}$.

- In $\mathcal{T}_\flat$ we define $[\Gamma] \equiv \{\Gamma' \mid \overline{\Gamma'} \equiv \Gamma\}$.

- We have three systems: $(\mathcal{T}, \to_\beta)$, $(\mathcal{T}, \to_{\beta\Pi})$ and $(\mathcal{T}_\flat, \to_\flat)$ with the axioms:
  - $(\lambda_{x:A}.B)C \to_\beta B[x := C]$
  - $(\flat_{x:A}.B)C \to_\flat B[x := C]$
  - $(\Pi_{x:A}.B)C \to_\Pi B[x := C]$
  - where $\to_{\beta\Pi} = \to_\beta \cup \to_\Pi$

- Church-Rosser: Let $r \in \{\beta, \beta\Pi, \flat\}$.
  If $B_1 \, {}_r\!\twoheadleftarrow A \twoheadrightarrow_r B_2$ then $\exists C$ such that $B_1 \twoheadrightarrow_r C \, {}_r\!\twoheadleftarrow B_2$.

The β-cube: R_β = type rules with 2 binders, $\rightarrow_\beta$ and (appl)

(axiom)	$\langle\rangle \vdash * : \square$
(start)	$$\frac{\Gamma \vdash A : s \quad x^s \notin \textsc{dom}\,(\Gamma)}{\Gamma, x^s{:}A \vdash x^s : A}$$
(weak)	$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad x^s \notin \textsc{dom}\,(\Gamma)}{\Gamma, x^s{:}C \vdash A : B}$$
(Π)	$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2 \quad (s_1, s_2) \in \boldsymbol{R}}{\Gamma \vdash \Pi_{x:A}.B : s_2}$$
(λ)	$$\frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash \Pi_{x:A}.B : s}{\Gamma \vdash \lambda_{x:A}.b : \Pi_{x:A}.B}$$
(conv$_\beta$)	$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$$
(appl)	$$\frac{\Gamma \vdash F : \Pi_{x:A}.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$$

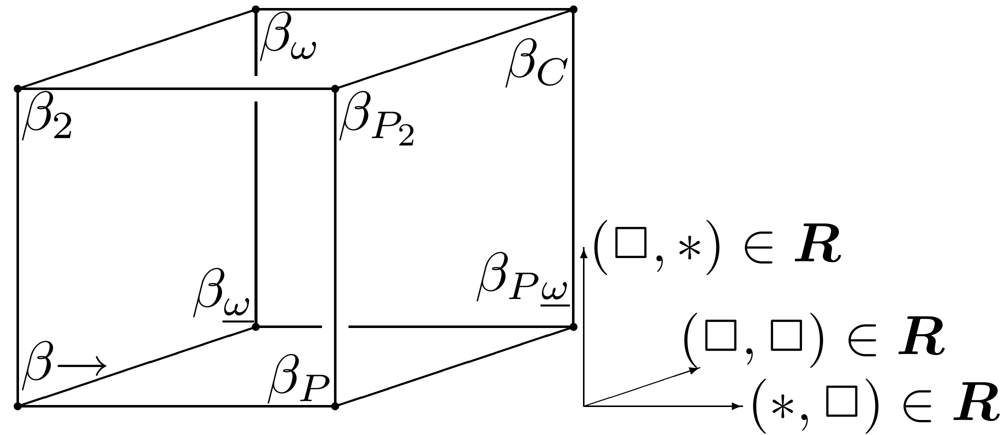


Figure 1: The β-cube of Barendregt

Our example in the system F of Girard

- If $x \notin FV(B)$ we write $A \to B$ instead of $\Pi_{x:A}.B$.
- $\alpha : *, f : \alpha \to \alpha \vdash \lambda_{x:\alpha}.f(f(x)) : \alpha \to \alpha : *$
  (we need the rule $(*, *)$).
- $\alpha : * \vdash \lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f(x)) : (\alpha \to \alpha) \to (\alpha \to \alpha) : *$
  (we need the rule $(*, *)$).
- $\vdash \lambda_{\alpha:*}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f(x)) : \Pi_{\alpha:*}.(\alpha \to \alpha) \to (\alpha \to \alpha) : *$
  (we need the rule $(\Box, *)$).

The π-cube: $R_\pi = R_\beta \backslash$ (conv$_\beta$) $\cup$ (conv$_{\beta\Pi}$) = Typing rules with two binders, $\rightarrow_{\beta\Pi}$ and (appl)

(axiom) (start) (weak) (Π) (λ) (appl)

(conv$_{\beta\Pi}$)
$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_{\beta\Pi} B'}{\Gamma \vdash A : B'}$$

The π_i-cube: $R_{\pi_i} = R_\pi \setminus$ (appl) $\cup$ (newappl) = typing rules with two binders, $\to_{\beta\Pi}$ and (newappl)

(axiom) (start) (weak) (Π) (λ)

(conv$_{\beta\Pi}$) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{\beta\Pi} B'}{\Gamma \vdash A : B'}$$

(newappl) $$\frac{\Gamma \vdash F : \Pi_{x:A}.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : (\Pi_{x:A}.B)a}$$

The $\flat$-cube: $R_\flat$ = Typing rules with one single binder $\to_\flat$ and (appl$\flat$)

(axiom) (start) (weak)

($\flat_1$) $$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2 \quad (s_1, s_2) \in \boldsymbol{R}}{\Gamma \vdash \flat_{x:A}.B : s_2}$$

($\flat_2$) $$\frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash \flat_{x:A}.B : s}{\Gamma \vdash \flat_{x:A}.b : \flat_{x:A}.B}$$

(conv$_\flat$) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\flat B'}{\Gamma \vdash A : B'}$$

(appl$\flat$) $$\frac{\Gamma \vdash F : \flat x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$$

$\flat_i$-cube: $R_{\flat_i} = R_\flat \backslash$ (appl$\flat$) $\cup$ (newappl$\flat$) = Typing rules with one single binder, $\to_\flat$ et (newappl$\flat$)

(axiom)	(start) (weak) ($\flat_1$) ($\flat_2$) (conv$_\flat$)
(newappl$\flat$)	$\dfrac{\Gamma \vdash F : \flat x{:}A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : (\flat x : A.B)a}$

6 desirable properties of a type system with reduction r

- *Types are correct (TC)*
  If $\Gamma \vdash A : B$ then $B \equiv \Box$ or $\Gamma \vdash B : s$ for $s \in \{*, \Box\}$.

- *Subject reduction (SR)* If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_r A'$ then $\Gamma \vdash A' : B$.

- *Preservation of types (PT)* If $\Gamma \vdash A : B$ and $B \twoheadrightarrow_r B'$ then $\Gamma \vdash A : B'$.

- *Strong Normalisation (SN)* If $\Gamma \vdash A : B$ then $\mathsf{SN}_{\rightarrow_r}(A)$ and $\mathsf{SN}_{\rightarrow_r}(B)$.

- *Subterms are typable (STT)* If A is $\vdash$-legal and if C is a sub-term of A then C is $\vdash$-legal.

- *Unicity of types*
  - *(UT1)* If $\Gamma \vdash A_1 : B_1$ and $\Gamma \vdash A_2 : B_2$ and $\Gamma \vdash A_1 =_r A_2$, then $\Gamma \vdash B_1 =_r B_2$.
  - *(UT2)* If $\Gamma \vdash B_1 : s$, $B_1 =_r B_2$ and $\Gamma \vdash A : B_2$ then $\Gamma \vdash B_2 : s$.

The correspondence between the β-, π- et π_i-cubes

- Lemma:
  - $\Gamma \vdash_\beta A : B$ iff $\Gamma \vdash_\pi A : B$
  - If $\Gamma \vdash_\beta A : B$ then $\Gamma \vdash_{\pi_i} A : B$.
  - If $\Gamma \vdash_{\pi_i} A : B$ then $\Gamma \vdash_\beta A : [B]_\Pi$
    where $[B]_\Pi$ is the Π-normal form of B.
- Lemma: The β-cube and the π-cube satisfy the six properties that are desirable for type systems.

The π_i-cube

- The π_i-cube loses three of its six properties
  Let $\Gamma = z : *, x : z$. We have that $\Gamma \vdash_{\pi_i} (\lambda_{y:z}.y)x : (\Pi_{y:z}.z)x$.
  - *We do not have TC* $(\Pi_{y:z}.z)x \not\equiv \Box$ and $\Gamma \not\vdash_{\pi_i} (\Pi_{y:z}.z)x : s$.
  - *We do not have SR* $(\lambda_{y:z}.y)x \rightarrow_{\beta\Pi} x$ but $\Gamma \not\vdash_{\pi_i} x : (\Pi_{y:z}.z)x$.
  - *We do not have UT2* $\vdash_{\pi_i} * : \Box$, $* =_{\beta\Pi} (\Pi_{z:*}.*)\alpha$, $\alpha : * \vdash_{\pi_i} (\lambda_{z:*}.*)\alpha : (\Pi_{z:*}.*)\alpha$ and $\not\vdash_{\pi_i} (\Pi_{z:*}.*)\alpha : \Box$

- But we have:
  - *We have UT1*
  - *We have STT*
  - *We have PT*
  - *We have SN*
  - *We have a weak form of TC* If $\Gamma \vdash_{\pi_i} A : B$ and *B does not have a Π-redexe* then either $B \equiv \Box$ or $\Gamma \vdash_{\pi_i} B : s$.
  - *We have a weak form of SR* If $\Gamma \vdash_{\pi_i} A : B$, *$B$ is not a Π-redex* and $A \twoheadrightarrow_{\beta\Pi} A'$ then $\Gamma \vdash_{\pi_i} A' : B$.

If we eliminate the distinction between λ and Π, it would still be possible to deduce the role of each binder

An algorithm which transformsa $\flat$-typing into a β-typing

Let $\Gamma \vdash_\flat A : B$. We define $(\Gamma \vdash_\flat A : B)^{-1} \in [\Gamma] \times [A] \times [B]$ by:

$$
\begin{array}{ll}
(\langle\rangle \vdash_\flat * : \Box)^{-1} & = (\langle\rangle, *, \Box) \\
(\Gamma, x^s : A \vdash_\flat * : \Box)^{-1} & = (\Gamma', x^s : A', *, \Box) \text{ where } (\Gamma \vdash_\flat A : s)^{-1} = (\Gamma', A', s) \\
(\Gamma \vdash_\flat x^s : C)^{-1} & = (\Gamma', x^s, C') \text{ where } (\Gamma \vdash_\flat C : s)^{-1} = (\Gamma', C', s) \\
(\Gamma \vdash_\flat \flat_{x:A}.B : C)^{-1} & = \begin{cases} (\Gamma', \Pi_{x:A'}.B', C') & \text{if } C =_\flat s_2 \text{ and i.} \\ (\Gamma', \lambda_{x:A'}.B', C') & \text{if } C =_\flat \flat_{x:A}.D \text{ and ii.} \end{cases} \\
(\Gamma \vdash_\flat Fa : C)^{-1} & = (\Gamma', F'a', C') \text{ where iii.}
\end{array}
$$

Where i, ii, and iii, are evident. We simply write the condition i. For the others see the article.

- i – $(\Gamma \vdash_\flat A : s_1)^{-1} = (\Gamma', A', s_1)$ for an s_1 such that $(s_1, s_2) \in \boldsymbol{R}$,
  - $(\Gamma, x : A \vdash_\flat B : s_2)^{-1} = (\Gamma'', x : A'', B', s_2)$
  - if $C \equiv s_2$ then $C' \equiv s_2$ otherwise
    if $C \not\equiv s_2$ then $(\Gamma \vdash_\flat C : s)^{-1} = (\Gamma''', C', s)$ for some s.

The isomorphism

- Theorem:
  1. if $\Gamma \vdash_\beta A : B$ then $\overline{\Gamma} \vdash_\flat \overline{A} : \overline{B}$.
  2. if $\Gamma \vdash_\flat A : B$ then
     - there is a unique $\Gamma' \in [\Gamma]$ such that Γ' is $\vdash_\beta$-legal, and
     - there is a unique $A' \in [A]$, a unique $B' \in [B]$ such that $\Gamma' \vdash_\beta A' : B'$.

     Moreoever, Γ', A' and B' are constructed by $^{-1}$ where $(\Gamma \vdash_\flat A : B)^{-1} = (\Gamma', A', B')$.
  3. The verification of types in the β-cube is equivalent to the verification of types in the $\flat$-cube

The $\flat$-cube

- The $\flat$-cube has five and a half properties:
  - *TC*
  - *SR*
  - *STT*
  - *PT*
  - *SN*
  - *UT2*
  - *Of course we do not have UT1 (and we don't even want it)*
    Using $(\Box, \Box)$: $\vdash_\beta \lambda_{x:*}.x : \Pi_{x:*}.*$ and $\vdash_\flat \flat_{x:*}.x : \flat_{x:*}.*$.

    Using $(\Box, *)$: $\vdash_\beta \Pi_{x:*}.x : *$ and $\vdash_\flat \flat_{x:*}.x : *$.
    Note that $\flat_{x:*}.* \neq_\flat *$.

We have an organised multiplicity of types in the $\flat$-cube

- Let $\mathsf{SN}_{\to_\flat}(B_1)$ and $\mathsf{SN}_{\to_\flat}(B_2)$. We say that $B_1 \stackrel{\diamond}{=}_\flat B_2$ iff $\mathrm{nf}_\flat(B_1) \equiv \flat^{i:1..n_1}_{x_i:F_i}.B$ and $\mathrm{nf}_\flat(B_2) \equiv \flat^{i:1..n_2}_{x_i:F_i}.B$, where $n_1, n_2 \geq 0$.
- *Lemma (MOT)* If $\Gamma \vdash_\flat A_1 : B_1$ and $\Gamma \vdash_\flat A_2 : B_2$ and $A_1 =_\flat A_2$, then $B_1 \stackrel{\diamond}{=}_\flat B_2$.
- Hence, the $\flat$-cube is elegant, and has all the desirable properties (we do not want UT1, MOT is enough).

Example

- Let $A \equiv \flat_{x_1:*}.\flat_{x_2:\flat_{y:C}.*}.\flat_{x_3:C}.\flat_{x_4:x_2x_3}.x_2x_3$

  Let $B \in \{\flat_{x_1:*}.\flat_{x_2:\flat_{y:C}.*}.\flat_{x_3:C}.\flat_{x_4:x_2x_3}.*,$
  $\flat_{x_1:*}.\flat_{x_2:\flat_{y:C}.*}.\flat_{x_3:C}.*,$
  $\flat_{x_1:*}.\flat_{x_2:\flat_{y:C}.*}.*,$
  $\flat_{x_1:*}.*,$
  $*\}$.
  We have that $\vdash_\flat A : B$

- In the β-cube we have:

$$
\begin{array}{llll}
\vdash_\beta \lambda_{x_1:*}.\lambda_{x_2:\Pi_{y:C}.*}.\lambda_{x_3:C}.\lambda_{x_4:x_2x_3}.x_2x_3 & : & \Pi_{x_1:*}.\Pi_{x_2:\Pi_{y:C}.*}.\Pi_{x_3:C}.\Pi_{x_4:x_2x_3}. & * \\
\vdash_\beta \lambda_{x_1:*}.\lambda_{x_2:\Pi_{y:C}.*}.\lambda_{x_3:C}.\Pi_{x_4:x_2x_3}.x_2x_3 & : & \Pi_{x_1:*}.\Pi_{x_2:\Pi_{y:C}.*}.\Pi_{x_3:C}. & * \\
\vdash_\beta \lambda_{x_1:*}.\lambda_{x_2:\Pi_{y:C}.*}.\Pi_{x_3:C}.\Pi_{x_4:x_2x_3}.x_2x_3 & : & \Pi_{x_1:*}.\Pi_{x_2:\Pi_{y:C}.*}. & * \\
\vdash_\beta \lambda_{x_1:*}.\Pi_{x_2:\Pi_{y:C}.*}.\Pi_{x_3:C}.\Pi_{x_4:x_2x_3}.x_2x_3 & : & \Pi_{x_1:*}. & * \\
\vdash_\beta \Pi_{x_1:*}.\Pi_{x_2:\Pi_{y:C}.*}.\Pi_{x_3:C}.\Pi_{x_4:x_2x_3}.x_2x_3 & : & & *
\end{array}
$$

  Note that only $\Pi_{x_1:*}.\Pi_{x_2:\Pi_{y:C}.*}.\Pi_{x_3:C}.\Pi_{x_4:x_2x_3}.x_2x_3$ can be a type. The other terms cannot.

Summary

- If the type system can do all that we want (as we have seen in the $\flat$-cube), why separate the levels of types and terms?
- Terms and types are not different. The only difference is the relation that exists between them. In $A : B$, the A on the left represents a term, the B on the right represents a type.
- Even if you do not want to use this system, its existence is already of significance.

Conclusion

- There are still millions of hidden gems in de Bruijn's Automath.
- Current work include de Bruijn's subtyping, de Bruijn's mathematical vernacular.
- Is it an exageration to say that: "if an idea in logical frameworks is good, it exists already in Automath. Look enough you will find it."
- Amazing that a single project (Automath) has resulted in such a treasure of influential ideas.
- Thank you de Bruijn, and thanks to your extremely modest, and extremely impressive x-PhD students/researchers (Rob Nederpelt, Bert van Benthem Jutting, Diedrick van Daalen, Herman Balsters, Jeff Zucker).
- What this group has done is truly impressive.

- I am particularly thankful to Rob Nederpelt who throughout decades has introduced me to the beautiful work in Automath and with whom I had my most extensive collaboration (and a very close and valuable friendship).

- Keep studying Automath, and keep discovering its beauty.

- Long live de Bruijn and long live Automath.

Bibliography

Abadi, Cardelli, Curien, and Levy. The $\lambda\sigma$ calculus. *Functional Programming*, 1991.

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pages 233–246, 1995.

Z.E.A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

Roel Bloo, Fairouz Kamareddine, and Rob Nederpelt. The Barendregt cube with definitions and generalised reduction. *Inform. & Comput.*, 126(2):123–143, May 1996.

N.G. de Bruijn. Generalising automath by means of a lambda-typed lambda calculus. 1984. Also in [Nederpelt et al., 1994], pages 313–337.

N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the chuirch rosser theorem. In *Indagationes Math*, pages 381–392. 1972. Also in [Nederpelt et al., 1994].

Philippe de Groote. The conservation theorem revisited. In *Proc. Int'l Conf. Typed Lambda Calculi and Applications*, pages 163–178. Springer, 1993.

F. Kamareddine and A. Ríos. A λ-calculus à la de bruijn with explicit substitutions. *Proceedings of Programming Languages Implementation and the Logic of Programs PLILP'95, Lecture Notes in Computer Science*, 982:45–62, 1995.

F. Kamareddine, R. Bloo, and R. Nederpelt. On Π-conversion in the λ-cube and the combination with abbreviations. *Ann. Pure Appl. Logic*, 97(1–3):27–45, 1999.

Fairouz Kamareddine. Postponement, conservation and preservation of strong normalisation for generalised reduction. *J. Logic Comput.*, 10(5):721–738, 2000.

Fairouz Kamareddine and Rob Nederpelt. Refining reduction in the λ-calculus. *J. Funct. Programming*, 5(4): 637–651, October 1995.

Fairouz Kamareddine and Rob Nederpelt. A useful λ-notation. *Theoret. Comput. Sci.*, 155(1):85–109, 1996.

Fairouz Kamareddine, Alejandro Ríos, and J. B. Wells. Calculi of generalised β-reduction and explicit substitutions: The type free and simply typed versions. *J. Funct. Logic Programming*, 1998(5), June 1998.

A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, pages 196–207, 1994. ISBN 0-89791-643-3.

A. J. Kfoury and J. B. Wells. New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi. In *Proc. 10th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 311–321, 1995. ISBN 0-8186-7050-9. URL `http://www.church-project.org/reports/electronic/Kfo+Wel:LICS-1995.pdf.gz`.

Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2):368–398, March 1994.

Rob Nederpelt. *Strong Normalization in a Typed Lambda Calculus With Lambda Structured Types*. PhD thesis, Eindhoven, 1973.

R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*. Studies in Logic and the Foundations of Mathematics **133**. North-Holland, Amsterdam, 1994.

Laurent Regnier. *Lambda calcul et réseaux*. PhD thesis, University Paris 7, 1992.