(Typed) λ-Calculi à la de Bruijn

Fairouz Kamareddine (Heriot-Watt University)

Tuesday 17 April 2014

De Bruijn's typed λ-calculi started with his Automath

- 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.

- Implementing extensive formal systems on the computer was never done before.

- De Bruijn, an extremely original mathematician, did every step his own way.

- He proudly announced at the ceremony of the publications of the collected Automath work: *I did it my way*.

- Dirk van Dalen said at the ceremony: *The Germans have their 3 B's, but we Dutch too have our 3 B's: Beth, Brouwer and de Bruijn*.

There is a fourth B:

Contraversy?

- In 1992, de Bruijn told me that when he announced his new project Automath at the start of January 1967, there was mixed reactions:
  - 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.

- It goes without saying that de Bruijn and his Automath shaped the way.
- De Bruijn's Automath influenced the Edinburgh Logical Frameworks.
- The Nuprl project has been connected to ideas in de Bruijn's Automath (e.g., telescopes).
- Coquand and Huet's calculus of constructions and consequently the proof checker Coq are 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 generalisations of typing rules influenced by 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).

- 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 review some details of de Bruijn's λ-calculus.

They look good together

Theme 1: De Bruijn Indices and Explicit Substitutions [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 $x, y, z, \ldots$).
- $(\lambda\lambda 2\ 1)2 \to_\beta (\lambda 2\ 1)[1 := 2] = \lambda(2[2 := 2])(1[2 := 2]) = \lambda 3\ 1$

Classical λ-calculus with de Bruijn indices

- Let $i, n \geq 1$ and $k \geq 0$
- $A \ ::= \ n \,|\, (\lambda B) \,|\, (BC)$
  $(\lambda A)B \to_\beta A\{\!\{1 \leftarrow B\}\!\}$
- $$\begin{aligned} U_k^i(AB) &= U_k^i(A)\, U_k^i(B) \\ U_k^i(\lambda A) &= \lambda(U_{k+1}^i(A)) \end{aligned} \qquad U_k^i(\mathtt{n}) = \begin{cases} \mathtt{n+i-1} & \text{if } \ n > k \\ \mathtt{n} & \text{if } \ n \leq k\,. \end{cases}$$
- $$\begin{aligned} (A_1 A_2)\{\!\{\mathtt{i} \leftarrow B\}\!\} &= (A_1\{\!\{\mathtt{i} \leftarrow B\}\!\})\,(A_2\{\!\{\mathtt{i} \leftarrow B\}\!\}) \\ (\lambda A)\{\!\{\mathtt{i} \leftarrow B\}\!\} &= \lambda(A\{\!\{\mathtt{i+1} \leftarrow B\}\!\}) \end{aligned}$$
  $$\mathtt{n}\{\!\{\mathtt{i} \leftarrow B\}\!\} = \begin{cases} \mathtt{n-1} & \text{if } \ n > i \\ U_0^i(B) & \text{if } \ n = i \\ \mathtt{n} & \text{if } \ n < i\,. \end{cases}$$
- 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 $\varphi^i_k A$.
- $A \ ::= \ n \mid (\lambda B) \mid (BC) \mid (A\sigma^i B) \mid (\varphi^i_k B) \qquad where \quad i, n \geq 1\,, \quad k \geq 0\,.$

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

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 was the first 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)	$\mathtt{n}\,[\uparrow]$	$\longrightarrow$	$\mathtt{n} + \mathtt{1}$
(Varshift2)	$\mathtt{n}\,[\uparrow \circ s]$	$\longrightarrow$	$\mathtt{n} + \mathtt{1}\,[s]$
(FVarCons)	$\mathtt{1}\,[a \cdot s]$	$\longrightarrow$	a
(RVarCons)	$\mathtt{n} + \mathtt{1}\,[a \cdot s]$	$\longrightarrow$	$\mathtt{n}\,[s]$
(FVarLift1)	$\mathtt{1}\,[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{1}$
(FVarLift2)	$\mathtt{1}\,[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{1}\,[t]$
(RVarLift1)	$\mathtt{n} + \mathtt{1}\,[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{n}[s \circ \uparrow]$
(RVarLift2)	$\mathtt{n} + \mathtt{1}\,[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{n}[s \circ (\uparrow \circ t)]$

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

$$\begin{array}{llcl}
(Map) & (a \cdot s) \circ t & \longrightarrow & a\,[t] \cdot (s \circ t) \\
(Ass) & (s \circ t) \circ u & \longrightarrow & s \circ (t \circ u) \\
(ShiftCons) & \uparrow \circ (a \cdot s) & \longrightarrow & s \\
(ShiftLift1) & \uparrow \circ \Uparrow(s) & \longrightarrow & s \circ \uparrow \\
(ShiftLift2) & \uparrow \circ (\Uparrow(s) \circ t) & \longrightarrow & s \circ (\uparrow \circ t) \\
(Lift1) & \Uparrow(s) \circ \Uparrow(t) & \longrightarrow & \Uparrow(s \circ t) \\
(Lift2) & \Uparrow(s) \circ (\Uparrow(t) \circ u) & \longrightarrow & \Uparrow(s \circ t) \circ u \\
(LiftEnv) & \Uparrow(s) \circ (a \cdot t) & \longrightarrow & a \cdot (s \circ t) \\
(IdL) & id \circ s & \longrightarrow & s \\
(IdR) & s \circ id & \longrightarrow & s \\
(LiftId) & \Uparrow(id) & \longrightarrow & id \\
(Id) & a\,[id] & \longrightarrow & a
\end{array}$$

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

A force in explicit substitutions à la $\lambda\sigma$

How is λse obtained from λs?

- They said, we can have *open terms (holes in proofs)* in $\lambda\sigma$, can you do so in λs?

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

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

- $(X\sigma^1 Y)\sigma^1 1$ 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).

- $(\sigma - \sigma)$:
  $A[x := B][y := C] = A[y := C][x := B[y := C]]$ if $x \neq y$ and $x \notin FV(C)$.

- $(\varphi - \sigma)$:
  $updated(A[x := B]) = (updatedA)[x := updatedB]$.

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:
$A\{\!\{\mathtt{i} \leftarrow B\}\!\}\{\!\{\mathtt{j} \leftarrow C\}\!\} = A\{\!\{\mathtt{j} + \mathtt{1} \leftarrow C\}\!\}\{\!\{\mathtt{i} \leftarrow B\{\!\{\mathtt{j} - \mathtt{i} + \mathtt{1} \leftarrow C\}\!\}\}\!\}$.

- 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 $l = 0$.

Where did the extra rules come from (continued)?

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

$(\varphi - \varphi\ 1)$ For $j \leq k+1$ we have: $U^i_{k+p}(U^j_p(A)) = U^j_p(U^i_{k+p+1-j}(A))$.

$(\varphi - \sigma)$ *[Distribution lemma]*
For $j \leq k+1$ we have: $U^i_k(A\{\!\{\mathsf{j} \leftarrow B\}\!\}) = U^i_{k+1}(A)\{\!\{\mathsf{j} \leftarrow U^i_{k+1-j}(B)\}\!\}$.

- $(\varphi - \varphi\ 1)$ with $p = 0$ is needed to prove $(\varphi - \sigma)$.

Everyone is happy. The center person is called "Maximum Happy"

Theme 2: 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.
- $(\lambda x.A)B \rightarrow_\beta A[x := B]$ becomes

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

- 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

Classical Notation	de Bruijn's Notation
$(\underline{(\lambda_x.(\lambda_y.\lambda_z.zd)c)b})a$	$\langle a\rangle\underline{\langle b\rangle[x]}\langle c\rangle[y][z]\langle d\rangle z$
$\downarrow_\beta$	$\downarrow_\beta$
$(\underline{(\lambda_y.\lambda_z.zd)c})a$	$\langle a\rangle\underline{\langle c\rangle[y]}[z]\langle d\rangle z$
$\downarrow_\beta$	$\downarrow_\beta$
$\underline{(\lambda_z.zd)a}$	$\underline{\langle a\rangle[z]}\langle d\rangle z$
$\downarrow_\beta$	$\downarrow_\beta$
ad	$\langle d\rangle a$

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

- This maks it easy to introduce local/global/mini reductions into the λ-calculus [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)	$((\lambda_x.\lambda_y.N)P)Q$ $\downarrow$ $(\lambda_x.N[y:=Q])P$	$\langle Q\rangle\langle P\rangle[x][y]N$ $\downarrow$ $\langle P\rangle[x][y:=Q]N$
(β_e)	? $\downarrow$?	$\langle Q\rangle\overline{s}[y]N$ $\downarrow$ $\overline{s}[y:=Q]N$

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; Kamareddine, 2000] 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., 1999a, 1998; Bloo et al., 1996] shows that they could reduce space/time needs.
- All these works have been heavily influenced by de Bruijn's Automath whose λ-notation facilitated the manipulation of redexes.
- All can be represented clearer in de Bruijn's notation.

An impressive thinker and entertainer

Claude warn out, glasses empty, eyeglasses somewhere

Even more: de Bruijn's generalised reduction has better properties

$$
\begin{array}{lll}
(\beta) & (\lambda_x.M)N \to M[x := N] & \\
(\beta_I) & (\lambda_x.M)N \to M[x := N] & \text{if } x \in FV(M) \\
(\beta_K) & (\lambda_x.M)N \to M & \text{if } x \notin FV(M) \\
(\theta) & (\lambda_x.N)PQ \to (\lambda_x.NQ)P & \\
(\beta_e) & (M)\overline{s}[x]N \to \overline{s}\{N[x := M] & \text{for } \overline{s} \text{ well-balanced.}
\end{array}
$$

- [Kamareddine, 2000] shows that β_e satisfies PSN, postponment of K-contraction and conservation (latter 2 properties fail for β-reduction).
- Conservation of β_e: If A is $\beta_e I$-normalisable then A is β_e-strongly normalisable.
- Postponment of K-contraction : Hence, discard arguments of K-redexes after I-reduction. This gives flexibility in implementation: unnecessary work can be delayed, or even completely avoided.

- Attempts have been made at establishing some reduction relations for which postponement of K-contractions and conservation hold.

- The picture is as follows (-N stands for normalising and $r \in \{\beta_I, \theta_K\}$).

(β_K-postponement for r)	If $M \to_{\beta_K} N \to_r O$ then $\exists P$ such that $M \twoheadrightarrow^{+}_{\beta_I \theta_K} P \twoheadrightarrow_{\beta_K}$	
(Conservation for β_I)	If M is β_I-N then M is β_I-SN	Barendregt's book
(Conservation for $\beta + \theta$)	If M is $\beta_I\theta_K$-N then M is β-SN	[de Groote, 1993]

- De Groote does not produce these results for a single reduction relation, but for $\beta + \theta$ (this is more restrictive than β_e).

- β_e is the first single relation to satisfy β_K-postponement and conservation.

- [Kamareddine, 2000] shows that:

(β_{eK}-postponement for β_e)	If $M \to_{\beta_{eK}} N \to_{\beta_{eI}} O$ then $\exists P$ such that $M \to_{\beta_{eI}} P \twoheadrightarrow^{+}_{\beta_e}$
(Conservation for β_e)	If M is β_{eI}-N then M is β_e-SN

De Bruijn does not tire. Look at Claude:

Never seen de Bruijn sleep on a train.

Nor sleep at a lecture

Compare with de Bruijn at a lecture

And here is Henk listening to de Bruijn's talk

This is de Bruijn at 9:15 am lecture my students at a short notice

Reduction Modulo versus Deduction Modulo

- Dowek, Hardin and Kirchner introduced a very neat and useful idea.

- Deduction modulo is a way to remove computational arguments from proofs by reasoning modulo a congruence on propositions.

- Separate computations and deductions in a clean way.

- Instead of $\dfrac{\Gamma \vdash P, \Delta \qquad \Gamma \vdash Q, \Delta}{\Gamma \vdash P \wedge Q, \Delta}$

- They write $\dfrac{\Gamma \vdash_c P, \Delta \qquad \Gamma \vdash_c Q, \Delta}{\Gamma \vdash_c R, \Delta}$ if R and $P \wedge Q$ are congruent ($R =_c P \wedge Q$)

Deduction modulo

- If $P =_c Q$ then
  $\Gamma \vdash_c P, \Delta$ iff $\Gamma \vdash_c Q, \Delta$ and
  $\Gamma, P \vdash_c \Delta$ iff $\Gamma, Q \vdash_c \Delta$
  and the proofs have the same size.

- For each congruence c, there is a set of axioms $\mathcal{T}$ such that
  $\mathcal{T}, \Gamma \vdash \Delta$ iff $\Gamma \vdash_c \Delta$

- Hence we have the same theorems in both formalisms but the proofs modulo are shorter because the standard deduction steps performing computations described by c are eliminated.

Reduction modulo

- To do reduction modulo, we need the notation of de Bruijn
- To give you an example why this notation is needed, we take the description of normal forms in a substitution calculus.
- The set of terms, noted Λs, of the λs-calculus is given as follows:

  $$\Lambda s ::= \mathsf{IN} \mid \Lambda s \Lambda s \mid \lambda \Lambda s \mid \Lambda s\, \sigma^i \Lambda s \mid \varphi^i_k \Lambda s \qquad where \quad i \geq 1\,, \;\; k \geq 0\,.$$

  The set of open terms, noted Λs_{op} is given as follows:

  $$\Lambda s_{op} ::= \mathbf{V} | \mathsf{IN} | \Lambda s_{op} \Lambda s_{op} | \lambda \Lambda s_{op} | \Lambda s_{op} \sigma \Lambda s_{op} | \varphi^i_k \Lambda s_{op} \qquad where \quad i \geq 1\,, \;\; k \geq 0$$

s_e-normal forms in classical notation

It is cumbersome to describe s_e-normal forms of open terms. But this description is needed to show the weak normalisation of the s_e-calculus. In classical notation, an open term is an s_e-normal form iff one of the following holds:

- $a \in \mathbf{V} \cup \mathbb{N}$, i.e. a is a variable or a de Bruijn number.
- $a = b\, c$, where b and c are s_e-normal forms.
- $a = \lambda b$, where b is an s_e-normal form.
- $a = b\, \sigma^j c$, where c is an s_e-nf and b is an s_e-nf of the form X, or $d\, \sigma^i e$ with $j < i$, or $\varphi^i_k d$ with $j \leq k$.
- $a = \varphi^i_k b$, where b is an s_e-nf of the form X, or $c\, \sigma^j d$ with $j > k+1$, or $\varphi^j_l c$ with $k < l$.

s_e-normal forms in item notation

- Write $a\,\sigma^i b = (b\,\sigma^i)a$ and $\varphi^i_k a = (\varphi^i_k)a$.
- We call a and b the bodies of these items.
- A *normal $\sigma\varphi$-segment $\overline{s}$* is a sequence of σ- and φ-items such that every pair of adjacent items in $\overline{s}$ are of the form:

$(\varphi^i_k)(\varphi^j_l)$ and $k < l$ $\qquad$ $(\varphi^i_k)(b\,\sigma^j)$ and $k < j - 1$

$(b\,\sigma^i)(c\,\sigma^j)$ and $i < j$ $\qquad$ $(b\,\sigma^j)(\varphi^i_k)$ and $j \le k$.

- The s_e-nf's can be described in item notation by the following syntax:

$$NF ::= \mathbf{V} \mid \mathsf{IN} \mid (NF\,\delta)NF \mid (\lambda)NF \mid \overline{s}\,\mathbf{V}$$

where $\overline{s}$ is a normal $\sigma\varphi$-segment whose bodies belong to NF.

Canonical Forms [Kamareddine et al., 2001b]

-

[ ] bachelors	( )[ ]-pairs	( )s bachelors	heart
$[x_1] \ldots [x_n]$	$(A_1)[y_1] \ldots (A_m)[y_m]$	$(B_1) \ldots (B_p)$	x

- Dans [Kfoury and Wells, 1995]

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

- For example, the canonical form of

$$[x][y](a)[z][x'](b)(c)(d)[y'][z'](e)x$$

is

$$[x][y][x'](a)[z](d)[y'](c)[z'](b)(e)x$$

How to get canonical forms?

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

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

$\to_\theta$ and $\to_\gamma$ are SN et CR. So θ-nf and γ-nf are unique.

$\theta(\gamma(A))$ and $\gamma(\theta(A))$ are the two *canonical*

Note that: $\theta(\gamma(A)) =_p \gamma(\theta(A))$ where $\to_p$ is the rule:

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

Reduction based on classes of canonical forms, Reduction Modulo

- Define $[A] = \{B \mid \theta(\gamma(A)) =_p \theta(\gamma(B))\}$.
- When $B \in [A]$, we write $B \approx_{\mathtt{equi}} A$.
- $\rightarrow_\theta, \rightarrow_\gamma, =_\gamma, =_\theta, =_p \subset \approx_{\mathtt{equi}} \subset =_\beta$ (strict inclusions).
- $A \rightsquigarrow_\beta B$ iff $\exists A' \in [A]. \exists B' \in [B]. A' \rightarrow_\beta B'$ (with compatibility)
- If $A \rightsquigarrow_\beta B$ then $\forall A' \in [A]. \forall B' \in [B]. A' \rightsquigarrow_\beta B'$.
- $\rightarrow_\beta \subset \rightarrow_g \subset \rightsquigarrow_\beta \subset =_\beta = \approx_\beta$.
- $\twoheadrightarrow_\beta$ *is CR:* Si $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$, then $\exists D$: $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.
- Let $r \in \{\rightarrow_\beta, \rightsquigarrow_\beta\}$. If $A \in SN_r$ and $A' \in [A]$ then $A' \in SN_r$.
- $A \in SN_{\rightsquigarrow_\beta}$ iff $A \in SN_{\rightarrow_\beta}$.

Properties of reduction modulo classes

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

Theme 3: Types and Functions à la de Bruijn

- *General definition of function* [Frege, 1879] is key to Frege's *formalisation of logic*.

- *Self-application of functions* was at the heart of *Russell's paradox* [Russell, 1902].

- To *avoid paradox* Russell controled function application via *type theory*.

- [Russell, 1903] gives the first type theory: the *Ramified Type Theory* (RTT).

- RTT is used in Principia Mathematica [Whitehead and Russell, 1910^1, 1927^2] 1910–1912.

- *Simple theory of types* (STT): [Ramsey, 1926], [Hilbert and Ackermann, 1928].

- Church's *simply typed λ-calculus* $\lambda\!\rightarrow$ 1940 = λ-calculus + STT.

- The hierarchies of types/orders of RTT and STT are *unsatisfactory*.
- The *notion of function adopted in the λ-calculus* is *unsatisfactory* (cf. [Kamareddine et al., 2003]).
- Hence, birth of *different systems of functions and types*, each with *different functional power*.
- Frege's functions $\neq$ Principia's functions $\neq$ *λ-calculus* functions (1932).
- Not all functions need to be *fully abstracted* as in the λ-calculus. For some functions, their values are enough.
- *Non-first-class functions* allow us to stay at a lower order (keeping decidability, typability, etc.) without losing the flexibility of the higher-order aspects.
- Non-first-class functions allow placing the type systems of modern theorem provers/programming languages like ML, LF and Automath more accurately in the modern hierarchy of types.

The evolution of functions with Frege, Russell and Church

- Historically, functions have long been treated as a kind of meta-objects.
- Function $values$ were the important part, not abstract functions.
- In the *low level/operational approach* there are only function values.
- The sine-function, is always expressed with a value: $\sin(\pi)$, $\sin(x)$ and properties like: $\sin(2x) = 2\sin(x)\cos(x)$.
- In many mathematics courses, one calls $f(x)$—and not f—the function.
- Frege, Russell and Church wrote $x \mapsto x + 3$ resp. as $x + 3$, $\hat{x} + 3$ and $\lambda x.x + 3$.
- Principia's *functions are based on Frege's Abstraction Principles* but can be first-class citizens. Frege used courses-of-values to speak about functions.
- Church made every function a first-class citizen. This is rigid and does not represent the development of logic in 20th century.

λ-calculus does not fully represent functionalisation

1. Abstraction from a subexpression $2 + 3 \mapsto x + 3$

2. Function construction $x + 3 \mapsto \lambda x.x + 3$

3. Application construction $(\lambda x.x + 3)2$

4. Concretisation to a subexpression $(\lambda x.(x + 3))2 \rightarrow 2 + 3$

- cannot abstract only half way: $x + 3$ is not a function, $\lambda x.x + 3$ is.

- cannot apply $x + 3$ to an argument: $(x + 3)2$ does not evaluate to 2+3.

Common features of modern types and functions

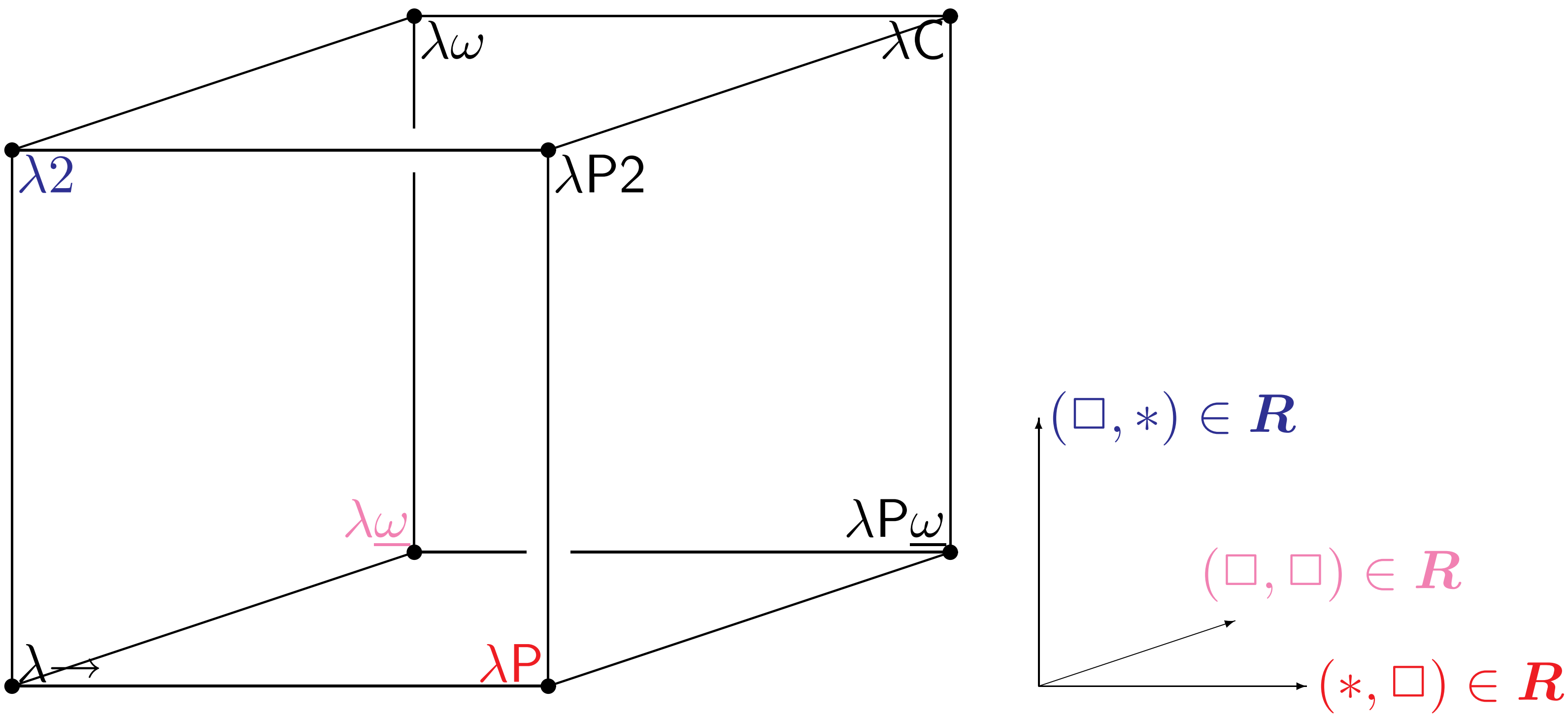
- We can *construct* a type by abstraction. (Write $A : *$ for A *is a type*)
  - $\lambda_{y:A}.y$, the identity over A *has type* $A \to A$
  - $\lambda_{A:*}.\lambda_{y:A}.y$, the polymorphic identity *has type* $\Pi_{A:*}.A \to A$
- We can *instantiate* types. E.g., if $A = \mathbb{N}$, then the identity over $\mathbb{N}$
  - $(\lambda_{y:A}.y)[A := \mathbb{N}]$ *has type* $(A \to A)[A := \mathbb{N}]$ or $\mathbb{N} \to \mathbb{N}$.
  - $(\lambda_{A:*}.\lambda_{y:A}.y)\mathbb{N}$ *has type* $(\Pi_{A:*}.A \to A)\mathbb{N} = (A \to A)[A := \mathbb{N}]$ or $\mathbb{N} \to \mathbb{N}$.
- $(\lambda x{:}\alpha.A)B \to_\beta A[x := B]$ $\qquad$ $(\Pi x{:}\alpha.A)B \to_\Pi A[x := B]$
- Write $A \to A$ as $\Pi_{y:A}.A$ when y not free in A.

The Barendregt Cube

- Syntax: $A ::= x \mid * \mid \Box \mid AB \mid \lambda x{:}A.B \mid \Pi x{:}A.B$
- Formation rule: $\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in \boldsymbol{R}$

	Simple	Poly-morphic	Depend-ent	Constr-uctors	Related system	Refs.
$\lambda\!\to$	$(*,*)$				λ^τ	[Church, 1940; B
$\lambda 2$	$(*,*)$	$(\Box,*)$			F	[Girard, 1972; Re
λP	$(*,*)$		$(*,\Box)$		AUT-QE, LF	[Bruijn, 1968; Ha
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$	POLYREC	[Renardel de Lava
λP2	$(*,*)$	$(\Box,*)$	$(*,\Box)$			[Longo and Mogg
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$	Fω	[Girard, 1972]
λP$\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$		
λC	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$	CC	[Coquand and Hu

The Barendregt Cube



Typing Polymorphic identity needs $(\Box, *)$

- $$\frac{y:*\vdash y:* \qquad y:*, x{:}y \vdash y:*}{y:*\vdash \Pi x{:}y.y:*}$$ by (Π) $(*, *)$
- $$\frac{y:*, x:y\vdash x:y \qquad y:*\vdash \Pi x{:}y.y:*}{y:*\vdash \lambda x:y.x : \Pi x{:}y.y}$$ by (λ)
- $$\frac{\vdash *:\Box \qquad y:*\vdash \Pi x{:}y.y:*}{\vdash \Pi y:*.\Pi x{:}y.y:*}$$ by (Π) $(\Box, *)$
- $$\frac{y:*\vdash \lambda x:y.x : \Pi x{:}y.y \qquad \vdash \Pi y:*.\Pi x{:}y.y:*}{\vdash \lambda y:*.\lambda x:y.x : \Pi y:*.\Pi x{:}y.y}$$ by (λ)

The story so far of the evolution of functions and types

- Functions have gone through a long process of evolution involving various degrees of abstraction/construction/instantiation/concretisation/evaluation.
- Types too have gone through a long process of evolution involving various degrees of abstraction/construction/instantiation/concretisation/evaluation.
- During their progress, some aspects have been added or removed.
- The development of types and functions have been interlinked and their abstraction/construction/instantiation/concretisation/evaluation have much in common.
- We also argue that some of the aspects that have been dismissed during their evolution need to be re-incorporated.

From the point of vue of ML

- The language ML is not based on all of system F (the second order polymorphic λ-calculus).
- This was not possible since it was not known then whether type checking and type finding are decidable.
- ML is based on a fragment of system F for which it was known that type checking and type finding are decidable.
- 23 years later after the design of ML, Wells showed that type checking and type finding in system F are undecidable.
- ML has polymorphism but not all the polymorphic power of system F.
- The question is, what system of functions and types does ML use?
- *A clean answer can be given when we re-incorporate the low-level function notion used by Frege and Russell (and de Bruijn) and dismissed by Church.*

- ML treats let val id $=$ (fn $x \Rightarrow x$) in (id id) end as this Cube term $(\lambda \mathsf{id}{:}(\Pi\alpha{:}*.\ \alpha \to \alpha).\ \mathsf{id}(\beta \to \beta)(\mathsf{id}\,\beta))(\lambda\alpha{:}*.\ \lambda x{:}\alpha.\ x)$

- To type this in the Cube, the $(\Box, *)$ rule is needed (i.e., $\lambda 2$).

- ML's typing rules forbid this expression:
  let val id $=$ (fn $x \Rightarrow x$) in (fn $y \Rightarrow y\,y$)(id id) end
  Its equivalent Cube term is this well-formed typable term of $\lambda 2$:
  $(\lambda \mathsf{id} : (\Pi\alpha{:}*.\ \alpha \to \alpha).$
  $\quad (\lambda y{:}(\Pi\alpha{:}*.\ \alpha \to \alpha).\ y(\beta \to \beta)(y\,\beta))$
  $\quad (\lambda\alpha{:}*.\ \mathsf{id}(\alpha \to \alpha)(\mathsf{id}\,\alpha)))$
  $(\lambda\alpha{:}*.\ \lambda x{:}\alpha.\ x)$

- Therefore, ML should not have the full Π-formation rule $(\Box, *)$.

- ML has limited access to the rule $(\Box, *)$.

- ML's type system is none of those of the eight systems of the Cube.

- [Kamareddine et al., 2001a] places the type system of ML (between $\lambda 2 + \lambda\underline{\omega}$).

LF

- LF [Harper et al., 1987] is often described as λP of the Barendregt Cube. However, Use of Π-formation rule $(*, \Box)$ is restricted in LF [Geuvers, 1993].

- We only need a type $\Pi x{:}A.B : \Box$ when PAT is applied during construction of the type $\Pi\alpha{:}\mathtt{prop}.*$ of the operator Prf where for a proposition Σ, Prf(Σ) is the type of proofs of Σ.

$$\frac{\mathtt{prop}{:}* \vdash \mathtt{prop}{:}\ * \qquad \mathtt{prop}{:}*, \alpha{:}\mathtt{prop} \vdash *{:}\Box}{\mathtt{prop}{:}* \vdash \Pi\alpha{:}\mathtt{prop}.* : \Box}.$$

- In LF, this is the only point where the Π-formation rule $(*, \Box)$ is used. But, Prf is only used when applied to $\Sigma{:}\mathtt{prop}$. We never use Prf on its own.

- This use is in fact based on a parametric constant rather than on Π-formation.

- Hence, the practical use of LF would not be restricted if we present Prf in a parametric form, and use $(*, \Box)$ as a parameter instead of a Π-formation rule.

- [Kamareddine et al., 2001a] precisely locate LF (between $\lambda\!\rightarrow$ and λP).

Parameters: What and Why

- We speak about *functions with parameters* when referring to functions with variable values in the *low-level* approach. The x in $f(x)$ is a parameter.

- Parameters enable the same expressive power as the high-level case, while allowing us to stay at a lower order. E.g. first-order with parameters versus second-order without [Laan and Franssen, 2001].

- Desirable properties of the lower order theory (decidability, easiness of calculations, typability) can be maintained, without losing the flexibility of the higher-order aspects.

- This low-level approach is still worthwhile for many exact disciplines. In fact, both in logic and in computer science it has certainly not been wiped out, and for good reasons.

Automath

- The first tool for mechanical representation and verification of mathematical proofs, AUTOMATH, has a parameter mechanism.
- Mathematical text in AUTOMATH written as a finite list of *lines* of the form:

  $x_1 : A_1, \ldots, x_n : A_n \vdash g(x_1, \ldots, x_n) = t : T$.

  Here g is a new name, an abbreviation for the expression t of type T and $x_1, \ldots, x_n$ are the parameters of g, with respective types $A_1, \ldots, A_n$.
- Each line introduces a new definition which is inherently parametrised by the variables occurring in the context needed for it.
- Developments of ordinary mathematical theory in AUTOMATH [Benthem Jutting, 1977] revealed that this combined definition and parameter mechanism is vital for keeping proofs manageable and sufficiently readable for humans.

Extending the Cube with parametric constants, see [Kamareddine et al., 2001a]

- We add parametric constants of the form $c(b_1, \dots, b_n)$ with $b_1, \dots, b_n$ terms of certain types and $c \in \mathcal{C}$.

- $b_1, \dots, b_n$ are called the *parameters* of $c(b_1, \dots, b_n)$.

- $\mathbf{R}$ allows several kinds of Π-constructs. We also use a set $\boldsymbol{P}$ of (s_1, s_2) where $s_1, s_2 \in \{*, \Box\}$ to allow several kinds of parametric constants.

- $(s_1, s_2) \in \boldsymbol{P}$ means that we allow parametric constants $c(b_1, \dots, b_n) : A$ where $b_1, \dots, b_n$ have types $B_1, \dots, B_n$ of sort s_1, and A is of type s_2.

- If both $(*, s_2) \in \boldsymbol{P}$ and $(\Box, s_2) \in \boldsymbol{P}$ then combinations of parameters allowed. For example, it is allowed that B_1 has type $*$, whilst B_2 has type $\Box$.

The Cube with parametric constants

- Let $(*,*) \subseteq \mathbf{R}, \boldsymbol{P} \subseteq \{(*,*),(*,\Box),(\Box,*),(\Box,\Box)\}$.

- $\lambda\mathbf{R}\boldsymbol{P} = \lambda\mathbf{R}$ and the two rules **($\vec{\textbf{C}}$-weak)** and **($\vec{\textbf{C}}$-app)**:

$$\frac{\Gamma \vdash b : B \qquad \Gamma, \Delta_i \vdash B_i : s_i \qquad \Gamma, \Delta \vdash A : s}{\Gamma, c(\Delta) : A \vdash b : B} \; (s_i, s) \in \boldsymbol{P}, c \text{ is } \Gamma\text{-fresh}$$

$$\frac{\begin{array}{lcll} \Gamma_1, c(\Delta){:}A, \Gamma_2 & \vdash & b_i{:}B_i[x_j{:=}b_j]_{j=1}^{i-1} & (i = 1, \ldots, n) \\ \Gamma_1, c(\Delta){:}A, \Gamma_2 & \vdash & A : s & (\text{if } n = 0) \end{array}}{\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash c(b_1, \ldots, b_n) : A[x_j{:=}b_j]_{j=1}^{n}}$$

$\Delta \equiv x_1{:}B_1, \ldots, x_n{:}B_n$.
$\Delta_i \equiv x_1{:}B_1, \ldots, x_{i-1}{:}B_{i-1}$

Properties of the Refined Cube

- (Correctness of types) If $\Gamma \vdash A : B$ then $(B \equiv \Box$ or $\Gamma \vdash B : S$ for some sort $S)$.

- (Subject Reduction SR) If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$

- (Strong Normalisation) For all $\vdash$-legal terms M, we have $\mathsf{SN}_{\twoheadrightarrow_\beta}(M)$.

- Other properties such as Uniqueness of types and typability of subterms hold.

- $\lambda\mathbf{R}\boldsymbol{P}$ is the system which has Π-formation rules $\boldsymbol{R}$ and parameter rules $\boldsymbol{P}$.

- Let $\lambda\mathbf{R}\boldsymbol{P}$ parametrically conservative (i.e., $(s_1, s_2) \in \boldsymbol{P}$ implies $(s_1, s_2) \in \boldsymbol{R}$).
  - The parameter-free system $\lambda\mathbf{R}$ is at least as powerful as $\lambda\mathbf{R}\boldsymbol{P}$.
  - If $\Gamma \vdash_{\mathbf{R}\boldsymbol{P}} a : A$ then $\{\Gamma\} \vdash_{\boldsymbol{R}} \{a\} : \{A\}$.

Example

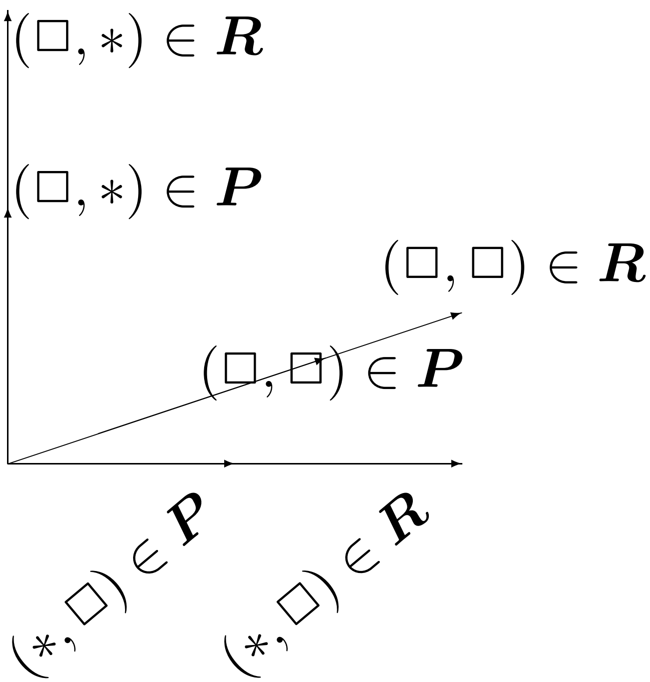
- $\boldsymbol{R} = \{(*,*),(*,\Box)\}$
  $\boldsymbol{P}_1 = \emptyset \qquad \boldsymbol{P}_2 = \{(*,*)\} \qquad \boldsymbol{P}_3 = \{(*,\Box)\} \qquad \boldsymbol{P}_4 = \{(*,*),(*,\Box)\}$
  All $\lambda\boldsymbol{R}\boldsymbol{P}_i$ for $1 \leq i \leq 4$ with the above specifications are all equal in power.

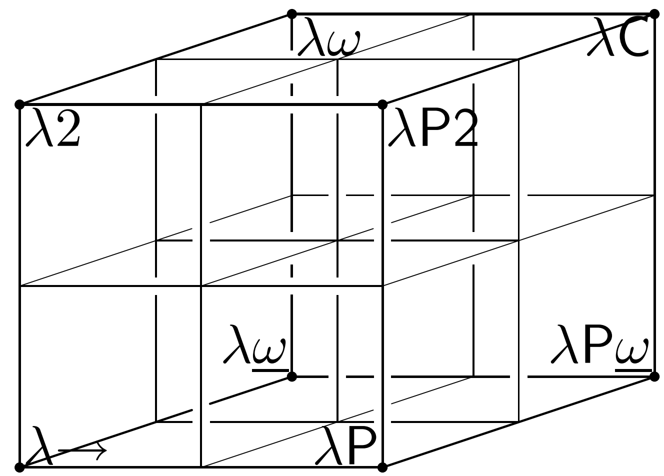
- $\boldsymbol{R}_5 = \{(*,*)\} \qquad \boldsymbol{P}_5 = \{(*,*),(*,\Box)\}$.
  $\lambda{\to} \ < \ \lambda\boldsymbol{R}_5\boldsymbol{P}_5 \ < \ \lambda$P: we can to talk about predicates:

$$\begin{array}{rcl}
\alpha & : & *, \\
\mathtt{eq}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha) & : & *, \\
\mathtt{refl}(\mathtt{x}{:}\alpha) & : & \mathtt{eq}(\mathtt{x},\mathtt{x}), \ . \\
\mathtt{symm}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha, \mathtt{p}{:}\mathtt{eq}(\mathtt{x},\mathtt{y})) & : & \mathtt{eq}(\mathtt{y},\mathtt{x}), \\
\mathtt{trans}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha, \mathtt{z}{:}\alpha, \mathtt{p}{:}\mathtt{eq}(\mathtt{x},\mathtt{y}), \mathtt{q}{:}\mathtt{eq}(\mathtt{y},\mathtt{z})) & : & \mathtt{eq}(\mathtt{x},\mathtt{z})
\end{array}$$

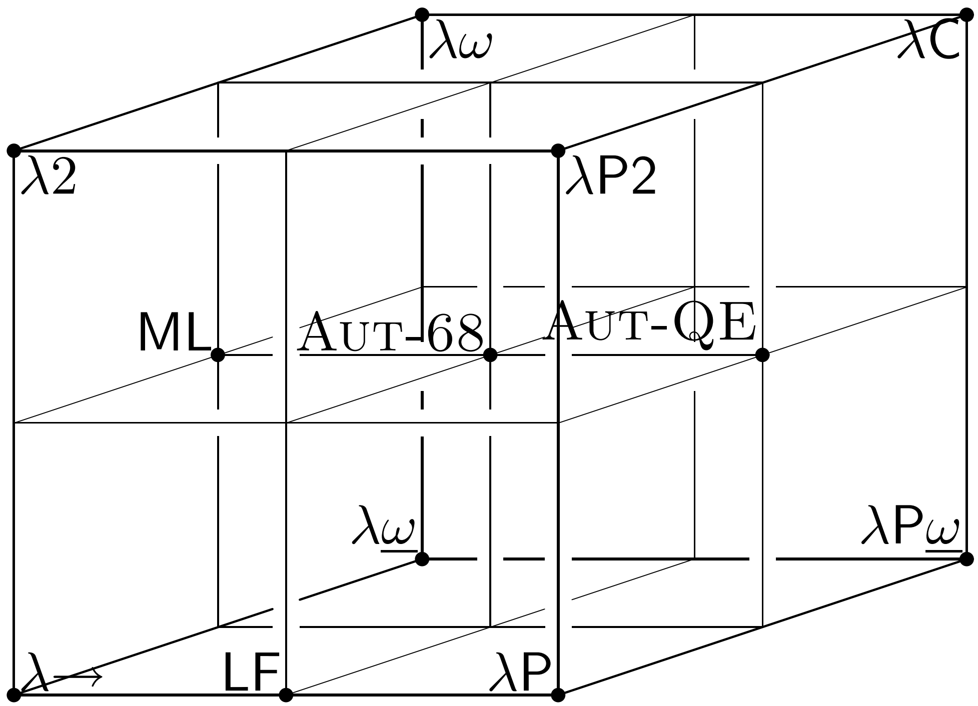
  eq not possible in $\lambda{\to}$.

The refined Barendregt Cube

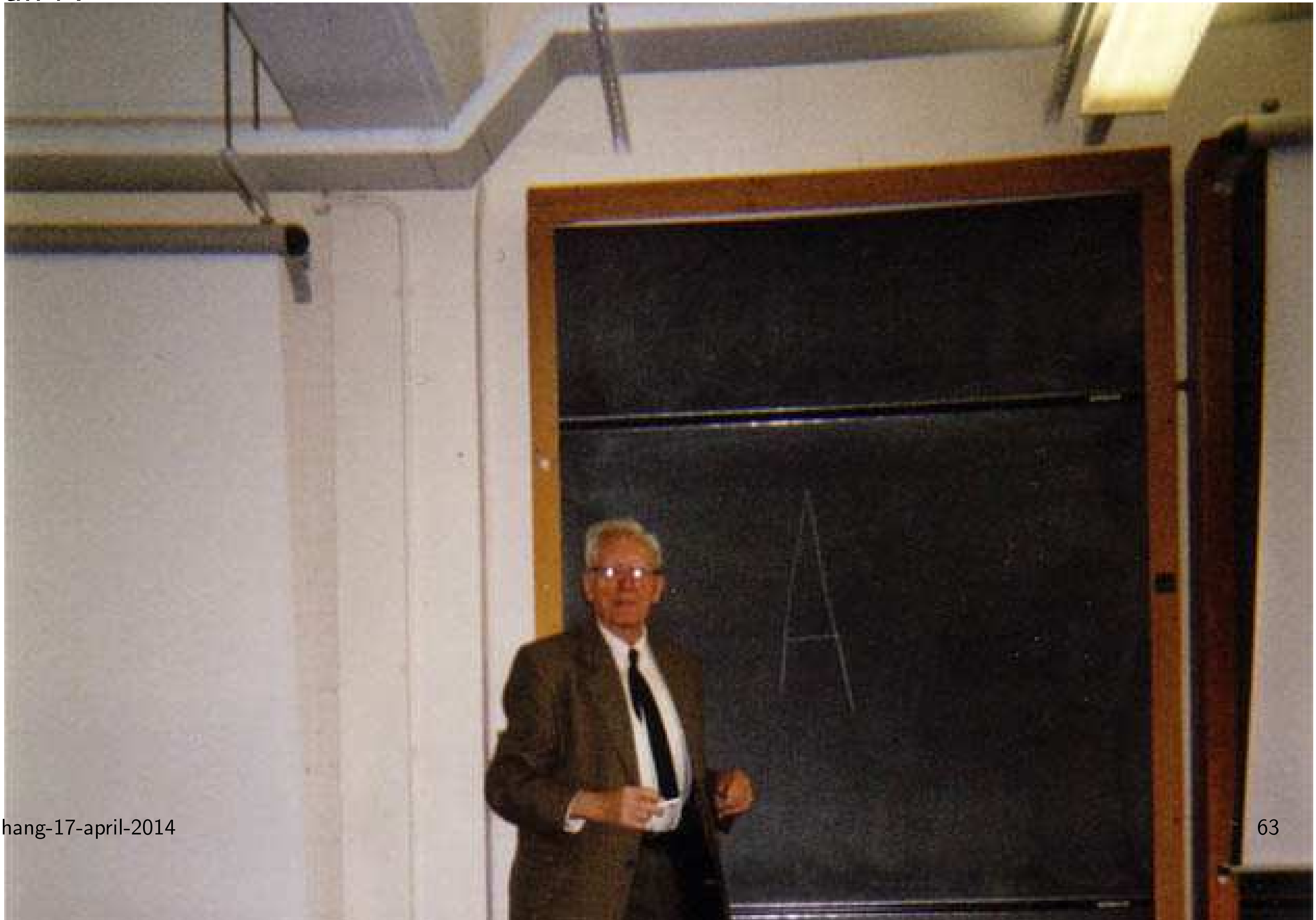




LF, ML, AUT-68, and AUT-QE in the refined Cube



And he was a great mind for whom even the most complex idea was as simple as an A



Theme 4: Identifying λ and Π (see [Kamareddine, 2005])

- In the cube of the generalised framework of type systems, we saw that the syntax for terms (functions) and types was intermixed with the only distinction being λ- versus Π-abstraction.

- We unify the two abstractions into one.
  $\mathcal{T}_\flat ::= \mathcal{V} \mid \boldsymbol{S} \mid \mathcal{T}_\flat\mathcal{T}_\flat \mid \flat\mathcal{V}{:}\mathcal{T}_\flat.\mathcal{T}_\flat$

- $\mathcal{V}$ is a set of variables and $\boldsymbol{S} = \{*, \Box\}$.

- The β-reduction rule becomes $(\flat)$ $\qquad (\flat_{x:A}.B)C \to_\flat B[x := C]$.

- Now we also have the old Π-reduction $(\Pi_{x:A}.B)C \to_\Pi B[x := C]$ which treats type instantiation like function instantiation.

- The type formation rule becomes

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

(axiom) $$\langle\rangle \vdash * : \Box$$

(start) $$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \; x \notin \textsc{dom}\,(\Gamma)$$

(weak) $$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B} \; x \notin \textsc{dom}\,(\Gamma)$$

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

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

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

Translations between the systems with 2 binders and those with one binder

- For $A \in \mathcal{T}$, we define $\overline{A} \in \mathcal{T}_\flat$ as follows:
  - $\overline{s} \equiv s \qquad \overline{x} \equiv x \qquad \overline{AB} \equiv \overline{A}\,\overline{B}$
  - $\overline{\lambda_{x:A}.B} \equiv \overline{\Pi_{x:A}.B} \equiv \flat_{x:\overline{A}}.\overline{B}$.
- For contexts we define: $\overline{\langle\rangle} \equiv \langle\rangle \quad \overline{\Gamma, x : A} \equiv \overline{\Gamma}, x : \overline{A}$.
- For $A \in \mathcal{T}_\flat$, we define $[A]$ to be $\{A' \in \mathcal{T}$ such that $\overline{A'} \equiv A\}$.
- For context, obviously: $[\Gamma] \equiv \{\Gamma'$ such that $\overline{\Gamma'} \equiv \Gamma\}$.

Isomorphism of the cube and the $\flat$-cube

- If $\Gamma \vdash A : B$ then $\overline{\Gamma} \vdash_\flat \overline{A} : \overline{B}$.
- If $\Gamma \vdash_\flat A : B$ then there are unique $\Gamma' \in [\Gamma]$, $A' \in [A]$ and $B' \in [B]$ such that $\Gamma' \vdash_\pi A' : B'$.
- *The $\flat$-cube enjoys all the properties of the cube except the unicity of types.*

Organised multiplicity of Types for $\vdash_\flat$ and $\to_\flat$ [Kamareddine, 2005]

For many type systems, unicity of types is not necessary (e.g. Nuprl).

We have however an organised multiplicity of types.

1. If $\Gamma \vdash_\flat A : B_1$ and $\Gamma \vdash_\flat A : B_2$, then $B_1 \overset{\diamond}{=}_\flat B_2$.
2. If $\Gamma \vdash_\flat A_1 : B_1$ and $\Gamma \vdash_\flat A_2 : B_2$ and $A_1 =_\flat A_2$, then $B_1 \overset{\diamond}{=}_\flat B_2$.
3. If $\Gamma \vdash_\flat B_1 : s_1$, $B_1 =_\flat B_2$ and $\Gamma \vdash_\flat A : B_2$ then $\Gamma \vdash_\flat B_2 : s_1$.
4. Assume $\Gamma \vdash_\flat A : B_1$ and $(\Gamma \vdash_\flat A : B_1)^{-1} = (\Gamma', A', B_1')$. Then $B_1 =_\flat B_2$ if:
   (a) either $\Gamma \vdash_\flat A : B_2$, $(\Gamma \vdash_\flat A : B_2)^{-1} = (\Gamma', A'', B_2')$ and $B_1' =_\beta B_2'$,
   (b) or $\Gamma \vdash_\flat C : B_2$, $(\Gamma \vdash_\flat C : B_2)^{-1} = (\Gamma', C', B_2')$ and $A' =_\beta C'$.

Theme 4: Extending the cube with Π-reduction loses subject reduction [Kamareddine et al., 1999b]

If we change (appl) by (new appl) in the cube we lose subject reduction.

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

$$\text{(new appl)}\quad \frac{\Gamma \vdash F : (\Pi_{x:A}.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : (\Pi_{x:A}.B)a}$$

[Kamareddine et al., 1999b] *solved the problem by re-incorporating Frege and Russell's notions of low level functions* (which was lost in Church's notion of function).

The same problem and solution can be repeated in our $\flat$-cube with type instantiation (Π-reduction).

Adding type instantiation to the typing rules of the $\flat$-cube

If we change (app$\flat$) by (new app$\flat$) in the $\flat$-cube we lose subject reduction.

$$(\mathsf{app}\flat) \quad \frac{\Gamma \vdash_\flat F : (\Pi_{x:A}.B) \qquad \Gamma \vdash_\flat a : A}{\Gamma \vdash_\flat Fa : B[x := a]}$$

$$(\mathsf{app}\flat\flat) \quad \frac{\Gamma \vdash_\flat F : (\flat_{x:A}.B) \qquad \Gamma \vdash_\flat a : A}{\Gamma \vdash_\flat Fa : (\flat_{x:A}.B)a}$$

Failure of correctness of types and subject reduction

- **Correctness of types no longer holds.** With (appl$\flat\flat$) one can have $\Gamma \vdash A : B$ without $B \equiv \Box$ or $\exists S\,.\,\Gamma \vdash B : S$.

- For example, $z : *, x : z \vdash (\flat_{y:z}.y)x : (\flat_{y:z}.z)x$ yet $(\flat_{y:z}.z)x \not\equiv \Box$ and $\forall s\,.\,z : *, x : z \not\vdash (\flat_{y:z}.z)x : s$.

- **Subject Reduction no longer holds.** That is, with (appl$\flat$): $\Gamma \vdash A : B$ and $A \twoheadrightarrow A'$ may not imply $\Gamma \vdash A' : B$.

- For example, $z : *, x : z \vdash (\flat_{y:z}.y)x : (\flat_{y:z}.z)x$ and $(\flat_{y:z}.y)x \to_\flat x$, but one can't show $z : *, x : z \vdash x : (\flat_{y:z}.z)x$.

Solving the problem

Keep all the typing rules of the $\flat$-cube the same except: replace (conv) by (new-conv), (appl$\flat$) by (appl$\flat\flat$) and add three new rules as follows:

(start-def) $$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : A}{\Gamma, x = B{:}A \vdash x : A} \qquad x \notin \mathrm{DOM}\,(\Gamma)$$

(weak-def) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad \Gamma \vdash D : C}{\Gamma, x = D{:}C \vdash A : B} \qquad x \notin \mathrm{DOM}\,(\Gamma)$$

(def) $$\frac{\Gamma, x = B{:}A \vdash C : D}{\Gamma \vdash (\flat x{:}A.C)B : D[x := B]}$$

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

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

In the conversion rule, $\Gamma \vdash B =_{def} B'$ is defined as:

- If $B =_\flat B'$ then $\Gamma \vdash B =_{def} B'$
- If $x = D : C \in \Gamma$ and B' arises from B by substituting one particular free occurrence of x in B by D then $\Gamma \vdash B =_{def} B'$.
- Our 3 new rules and the definition of $\Gamma \vdash B =_{def} B'$ are trying to re-incorporate low-level aspects of functions that are not present in Church's λ-calculus.
- *In fact, our new framework is closer to Frege's abstraction principle and the principles $*$9·14 and $*$9·15 of [Whitehead and Russell, 1910^1, 1927^2].*

Correctness of types holds.

- We demonstrate this with the earlier example.
- Recall that we have $z:*, x:z \vdash (\flat_{y:z}.y)x : (\flat_{y:z}.z)x$ and want that for some s, $z:*, x:z \vdash (\flat_{y:z}.z)x : s$.
- Here is how the latter formula now holds:

$$\begin{array}{ll} z:*, x:z \vdash z:* & \text{(start and weakening)} \\ z:*, x:z.y=x:z \vdash z:* & \text{(weakening)} \\ z:*, x:z \vdash (\flat_{y:z}.z)x : *[y:=x] \equiv * & \text{(def rule)} \end{array}$$

Subject Reduction holds.

- We demonstrate this with the earlier example.

- Recall that we have $z : *, x : z \vdash (\flat_{y:z}.y)x : (\flat_{y:z}.z)x$ and $(\lambda_{y:z}.y)x \to_\beta x$ and we need to show that $z : *, x : z \vdash x : (\flat_{y:z}.z)x$.

- Here is how the latter formula now holds:

$$\begin{array}{lll} a. & z : *, x : z \vdash x : z & \text{(start and weakening)} \\ b. & z : *, x : z \vdash (\flat_{y:z}.z)x : * & \text{(from 1 above)} \\ & z : *, x : z \vdash x : (\flat_{y:z}.z)x & \text{(conversion, } a\text{, } b\text{, and } z =_\beta (\flat_{y:z}.z)x\text{)} \end{array}$$

He liked to join in everything

He was a great listener

He was an impressive company

He was a gentleman

©Gérard Huet

Bibliography

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.

H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, revised edition, 1984.

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.

L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system*. PhD thesis, Eindhoven University of Technology, 1977. Published as Mathematical Centre Tracts nr. 83 (Amsterdam, Mathematisch Centrum, 1979).

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. The mathematical language AUTOMATH, its usage and some of its extensions. In M. Laudet, D. Lacombe, and M. Schuetzenberger, editors, *Symposium on Automatic Demonstration*, pages 29–61, IRIA, Versailles, 1968. Springer Verlag, Berlin, 1970. Lecture Notes in Mathematics **125**; also in [Nederpelt et al., 1994], pages 73–100.

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

A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

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.

G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Nebert, Halle, 1879. Also in [Heijenoort, 1967], pages 1–82.

J.H. Geuvers. *Logics and Type Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings Second Symposium on Logic in Computer Science*, pages 194–204, Washington D.C., 1987. IEEE.

J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, Massachusetts, 1967.

D. Hilbert and W. Ackermann. *Grundzüge der Theoretischen Logik*. Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen, Band XXVII. Springer Verlag, Berlin, first edition, 1928.

J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

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, 1999a.

F. Kamareddine, L. Laan, and R.P. Nederpelt. Refining the Barendregt cube using parameters. In *Proceedings of the Fifth International Symposium on Functional and Logic Programming, FLOPS 2001*, pages 375–389, 2001a.

F. Kamareddine, T. Laan, and R. Nederpelt. Revisiting the notion of function. *Logic and Algebraic programming*, 54:65–107, 2003.

Fairouz Kamareddine. Typed lambda-calculi with one binder. *J. Funct. Program.*, 15(5):771–796, 2005.

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.

Fairouz Kamareddine, Roel Bloo, and Rob Nederpelt. On pi-conversion in the lambda-cube and the combination with abbreviations. *Ann. Pure Appl. Logic*, 97(1-3):27–45, 1999b.

Fairouz Kamareddine, Roel Bloo, and Rob Nederpelt. De Bruijn's syntax and reductional equivalence of lambda terms. In *Proc. 3rd Int'l Conf. Principles & Practice Declarative Programming*, 5–7 September 2001b. ISBN 1-58113-388-X.

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.

Twan Laan and Michael Franssen. Parameters for first order logic. *Logic and Computation*, 2001.

G. Longo and E. Moggi. Constructive natural deduction and its modest interpretation. Technical Report CMU-CS-88-131, Carnegie Mellono University, Pittsburgh, USA, 1988.

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.

F.P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society,* 2nd series, 25: 338–384, 1926.

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

G.R. Renardel de Lavalette. Strictness analysis via abstract interpretation for recursively defined types. *Information and Computation*, 99:154–177, 1991.

J.C. Reynolds. *Towards a theory of type structure*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

B. Russell. Letter to Frege. English translation in [Heijenoort, 1967], pages 124–125, 1902.

B. Russell. *The Principles of Mathematics*. Allen & Unwin, London, 1903.

A.N. Whitehead and B. Russell. *Principia Mathematica*, volume I, II, III. Cambridge University Press, 1910^1, 1927^2. All references are to the first volume, unless otherwise stated.