De Bruijn’s typed $\lambda$-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 languages (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 genius ideas of de Bruijn in his Automath project.

• In this talk, I will review some details of de Bruijn’s $\lambda$-calculus.
They look good together
Theme 1: De Bruijn Indices and Explicit Substitutions
[de Bruijn, 1972]

- Classical $\lambda$-calculus:
  \[
  A ::= x \mid (\lambda x. B) \mid (BC) \\
  (\lambda x. A) B \rightarrow_\beta A[x := B]
  \]

- $(\lambda x. \lambda y. xy)y \rightarrow_\beta (\lambda y. xy)[x := y] \neq \lambda y. yy$

- $(\lambda x. \lambda y. xy)y \rightarrow_\beta (\lambda y. xy)[x := y] =_\alpha (\lambda z. xz)[x := y] = \lambda z. yz$

- $\lambda x. x$ and $\lambda y. y$ are the same function. Write this function as $\lambda 1$.

- Assume a free variable list (say $x, y, z, \ldots$).

- $(\lambda \lambda 2 \ 1)2 \rightarrow_\beta (\lambda 2 \ 1)[1 := 2] = \lambda(2[2 := 2])(1[2 := 2]) = \lambda 3 \ 1$
Classical $\lambda$-calculus with de Bruijn indices

- Let $i, n \geq 1$ and $k \geq 0$

- $A ::= n \mid (\lambda B) \mid (BC)$
  $(\lambda A)B \rightarrow_\beta A\{1 \leftarrow B\}$

- $U^i_k(AB) = U^i_k(A) U^i_k(B)$
  $U^i_k(\lambda A) = \lambda(U^i_{k+1}(A))$

- $U^i_k(n) = \begin{cases} n + i - 1 & \text{if } n > k \\ n & \text{if } n \leq k \end{cases}$

- $(A_1 A_2)\{i \leftarrow B\} = (A_1\{i \leftarrow B\})(A_2\{i \leftarrow B\})$
  $(\lambda A)\{i \leftarrow B\} = \lambda(A\{i + 1 \leftarrow B\})$

- $n\{i \leftarrow B\} = \begin{cases} n - 1 & \text{if } n > i \\ U^i_0(B) & \text{if } n = i \\ n & \text{if } n < i \end{cases}$

- Numerous implementations of proof checkers and programming languages have been based on de Bruijn indices.
From classical $\lambda$-calculus with de Bruijn indices to substitution calculus $\lambda s$ [Kamareddine and Ríos, 1995]

- Write $A\{n \leftarrow B\}$ as $A\sigma^n B$ and $U^i_k(A)$ as $\varphi^i_k A$.

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma$-generation</td>
<td>$(\lambda A) B$</td>
<td>$A \sigma^1 B$</td>
</tr>
<tr>
<td>$\sigma$-$\lambda$-transition</td>
<td>$(\lambda A) \sigma^i B$</td>
<td>$\lambda(A \sigma^{i+1} B)$</td>
</tr>
<tr>
<td>$\sigma$-app-transition</td>
<td>$(A_1 A_2) \sigma^i B$</td>
<td>$(A_1 \sigma^i B) (A_2 \sigma^i B)$</td>
</tr>
</tbody>
</table>
| $\sigma$-destruction | $n \sigma^i B$ | \[\begin{cases} 
    n - 1 & \text{if } n > i \\
    \varphi^i_0 B & \text{if } n = i \\
    n & \text{if } n < i 
\end{cases}\] |
| $\varphi$-$\lambda$-transition | $\varphi^i_k (\lambda A)$ | $\lambda(\varphi^{i+1}_k A)$ |
| $\varphi$-app-transition | $\varphi^i_k (A_1 A_2)$ | $(\varphi^i_k A_1) (\varphi^i_k A_2)$ |
| $\varphi$-destruction | $\varphi^i_k n$ | \[\begin{cases} 
    n + i - 1 & \text{if } n > k \\
    n & \text{if } n \leq k 
\end{cases}\] |
1. The $s$-calculus (i.e., $\lambda s$ minus $\sigma$-generation) is strongly normalising,

2. The $\lambda s$-calculus is confluent and simulates (in small steps) $\beta$-reduction

3. The $\lambda s$-calculus preserves strong normalisation PSN.

4. The $\lambda s$-calculus has a confluent extension with open terms $\lambda se$.

- The $\lambda s$-calculus was the first calculus of substitutions which satisfies all the above properties 1., 2., 3. and 4.
\( \lambda \nu \) [Benaissa et al., 1996]

Terms:
\[
\Lambda \nu^t ::= \text{IN} | \Lambda \nu^t \Lambda \nu^t | \lambda \Lambda \nu^t | \Lambda \nu^t[\Lambda \nu^s]
\]

Substitutions:
\[
\Lambda \nu^s ::= \uparrow | \uparrow (\Lambda \nu^s) | \Lambda \nu^t.
\]

\begin{align*}
(Beta) & \quad (\lambda a) b \rightarrow a[b/] \\
(App)  & \quad (a b)[s] \rightarrow (a[s])(b[s]) \\
(Abs)  & \quad (\lambda a)[s] \rightarrow \lambda(a[\uparrow(s)]) \\
(FVar) & \quad 1[a/] \rightarrow a \\
(RVar) & \quad n + 1[a/] \rightarrow n \\
(FVarLift) & \quad 1[\uparrow(s)] \rightarrow 1 \\
(RVarLift) & \quad n + 1[\uparrow(s)] \rightarrow n[s][\uparrow] \\
(VarShift) & \quad n[\uparrow] \rightarrow n + 1
\end{align*}

\( \lambda \nu \) satisfies 1., 2., and 3., but does not have a confluent extension on open terms.
Terms: \( \Lambda \sigma^t \ ::= \text{IN} | \Lambda \sigma^t \Lambda \sigma^t | \lambda \Lambda \sigma^t [\Lambda \sigma^s] \)

Substitutions: \( \Lambda \sigma^s \ ::= \text{id} | \uparrow | \uparrow (\Lambda \sigma^s) | \Lambda \sigma^t \cdot \Lambda \sigma^s | \Lambda \sigma^s \circ \Lambda \sigma^s \).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Beta)</td>
<td>( (\lambda a) b )</td>
<td>( a [b \cdot \text{id}] )</td>
</tr>
<tr>
<td>(App)</td>
<td>( (a b)[s] )</td>
<td>( (a [s]) (b [s]) )</td>
</tr>
<tr>
<td>(Abs)</td>
<td>( (\lambda a)[s] )</td>
<td>( \lambda(a [\uparrow(s)]) )</td>
</tr>
<tr>
<td>(Clos)</td>
<td>( (a [s])[t] )</td>
<td>( a [s \circ t] )</td>
</tr>
<tr>
<td>(Varshift1)</td>
<td>( n[\uparrow] )</td>
<td>( n + 1 )</td>
</tr>
<tr>
<td>(Varshift2)</td>
<td>( n[\uparrow \circ s] )</td>
<td>( n + 1 [s] )</td>
</tr>
<tr>
<td>(FVarCons)</td>
<td>( 1[a \cdot s] )</td>
<td>( a )</td>
</tr>
<tr>
<td>(RVarCons)</td>
<td>( n + 1[a \cdot s] )</td>
<td>( n [s] )</td>
</tr>
<tr>
<td>(FVarLift1)</td>
<td>( 1[\uparrow (s)] )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>(FVarLift2)</td>
<td>( 1[\uparrow (s) \circ t] )</td>
<td>( 1 [t] )</td>
</tr>
<tr>
<td>(RVarLift1)</td>
<td>( n + 1[\uparrow (s)] )</td>
<td>( n[s \circ \uparrow] )</td>
</tr>
<tr>
<td>(RVarLift2)</td>
<td>( n + 1[\uparrow (s) \circ t] )</td>
<td>( n[s \circ (\uparrow \circ t)] )</td>
</tr>
</tbody>
</table>
\( \lambda \sigma \downarrow \) rules continued

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Map)</td>
<td>((a \cdot s) \circ t)</td>
<td>(a [t] \cdot (s \circ t))</td>
</tr>
<tr>
<td>(Ass)</td>
<td>((s \circ t) \circ u)</td>
<td>(s \circ (t \circ u))</td>
</tr>
<tr>
<td>(ShiftCons)</td>
<td>(\uparrow \circ (a \cdot s))</td>
<td>(s)</td>
</tr>
<tr>
<td>(ShiftLift1)</td>
<td>(\uparrow \circ \uparrow (s))</td>
<td>(s \circ \uparrow)</td>
</tr>
<tr>
<td>(ShiftLift2)</td>
<td>(\uparrow \circ (\uparrow (s) \circ t))</td>
<td>(s \circ (\uparrow \circ t))</td>
</tr>
<tr>
<td>(Lift1)</td>
<td>(\uparrow (s) \circ \uparrow (t))</td>
<td>(\uparrow (s \circ t))</td>
</tr>
<tr>
<td>(Lift2)</td>
<td>(\uparrow (s) \circ (\uparrow (t) \circ u))</td>
<td>(\uparrow (s \circ t) \circ u)</td>
</tr>
<tr>
<td>(LiftEnv)</td>
<td>(\uparrow (s) \circ (a \cdot t))</td>
<td>(a \cdot (s \circ t))</td>
</tr>
<tr>
<td>(IdL)</td>
<td>(id \circ s)</td>
<td>(s)</td>
</tr>
<tr>
<td>(IdR)</td>
<td>(s \circ id)</td>
<td>(s)</td>
</tr>
<tr>
<td>(LiftId)</td>
<td>(\uparrow (id))</td>
<td>(id)</td>
</tr>
<tr>
<td>(Id)</td>
<td>(a [id])</td>
<td>(a)</td>
</tr>
</tbody>
</table>

\( \lambda \sigma \downarrow \) satisfies 1., 2., and 4., but does not have PSN.
A force in explicit substitutions à la $\lambda\sigma$
How is $\lambda se$ obtained from $\lambda s$?

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

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

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

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

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

- Simple: add de Bruijn’s metasubstitution and distribution lemmas to the rules of $\lambda s$: 

Beihang-17-april-2014
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).

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

• ($\varphi - \sigma$):
  \[ \text{updated}(A[x := B]) = (\text{updated}A)[x := \text{updated}B]. \]
Where did the extra rules come from?

In de Bruijn’s classical $\lambda$-calculus we have the lemmas:

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

$(\varphi - \varphi 2)$ 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: $U_k^i(A)\{j \leftarrow B\} = U_k^i(A\{j - i + 1 \leftarrow B\})$.

$(\sigma - \sigma)$ [Meta-substitution lemma] For $i \leq j$ we have:

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

- 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 $l = 0$. 

Beihang-17-april-2014
Where did the extra rules come from (continued)?

In de Bruijn’s classical $\lambda$-calculus we also have the lemmas:

$(\varphi - \varphi_1)$ 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: $U_k^i(A\{j \leftarrow B\}) = U_{k+1}^i(A)\{j \leftarrow U_{k+1-j}^i(B)\}$.

- $(\varphi - \varphi_1)$ 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

- $\mathcal{I}(x) = x$, $\mathcal{I}(\lambda x. B) = [x] \mathcal{I}(B)$, $\mathcal{I}(AB) = \langle \mathcal{I}(B) \rangle \mathcal{I}(A)$

- $(\lambda x. \lambda y. xy) z$ 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

\[(\lambda_x. (\lambda_y. \lambda_z. zd) c) b) a\]

\[\downarrow_{\beta}\]

\[(\lambda_y. \lambda_z. zd) c) a\]

\[\downarrow_{\beta}\]

\[(\lambda_z. zd) a\]

\[\downarrow_{\beta}\]

\[ad\]

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

\[\downarrow_{\beta}\]

\[\langle a \rangle \langle c \rangle [y] [z] \langle d \rangle z\]

\[\downarrow_{\beta}\]

\[\langle a \rangle [z] \langle d \rangle z\]

\[\downarrow_{\beta}\]

\[\langle d \rangle a\]

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

• This makes it easy to introduce local/global/mini reductions into the $\lambda$-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

<table>
<thead>
<tr>
<th>Name</th>
<th>In Classical Notation</th>
<th>In de Bruijn’s notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\theta)$</td>
<td>$((\lambda_x.N)P)Q$</td>
<td>$\langle Q\rangle \langle P \rangle [x]N \downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td></td>
<td>$(\lambda_x.NQ)P$</td>
<td>$\langle P \rangle [x] \langle Q \rangle N$</td>
</tr>
<tr>
<td>$(\gamma)$</td>
<td>$(\lambda_x.\lambda_y.N)P$</td>
<td>$\langle P \rangle [x][y]N \downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\lambda_y.(\lambda_x.N)P$</td>
<td>$\langle Q \rangle [y] \langle P \rangle [x]N$</td>
</tr>
<tr>
<td>$(\gamma_C)$</td>
<td>$((\lambda_x.\lambda_y.N)P)Q$</td>
<td>$\langle Q\rangle \langle P \rangle [x][y]N \downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td></td>
<td>$(\lambda_y.(\lambda_x.N)P)Q$</td>
<td>$\langle Q \rangle [y] \langle P \rangle [x]N$</td>
</tr>
<tr>
<td>$(\eta)$</td>
<td>$((\lambda_x.\lambda_y.N)P)Q$</td>
<td>$\langle Q\rangle \langle P \rangle [x][y]N \downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td></td>
<td>$(\lambda_x.N[y := Q])P$</td>
<td>$\langle P \rangle [x][y := Q]N$</td>
</tr>
<tr>
<td>$(\beta_e)$</td>
<td>$?$</td>
<td>$\langle Q \rangle \overline{s}[y]N \downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td></td>
<td>$?$</td>
<td>$\overline{s}[y := Q]N$</td>
</tr>
</tbody>
</table>
A Few Uses of these reductions/term reshuffling

- Regnier [1992] uses $\theta$ and $\gamma$ 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 $\lambda$-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

\[(\beta) \quad (\lambda_x. M) N \rightarrow M[x := N]\]

\[(\beta_I) \quad (\lambda_x. M) N \rightarrow M[x := N] \quad \text{if} \ x \in \text{FV}(M)\]

\[(\beta_K) \quad (\lambda_x. M) N \rightarrow M \quad \text{if} \ x \notin \text{FV}(M)\]

\[(\theta) \quad (\lambda_x. N) PQ \rightarrow (\lambda_x. NQ) P\]

\[(\beta_e) \quad (M)\overline{s}[x] N \rightarrow \overline{s}\{N[x := M]\} \quad \text{for} \ \overline{s} \ \text{well-balanced.}\]

- [Kamareddine, 2000] shows that \(\beta_e\) satisfies PSN, postponment of \(K\)-contraction and conservation (latter 2 properties fail for \(\beta\)-reduction).

- Conservation of \(\beta_e\): If \(A\) is \(\beta_e I\)-normalisable then \(A\) is \(\beta_e\)-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 $r \in \{\beta_I, \theta_K\}$).

\[
\begin{align*}
(\beta_K\text{-postponement for } r) & \quad \text{If } M \rightarrow_{\beta_K} N \rightarrow_{r} O \text{ then } \exists P \text{ such that } M \rightarrow_{\beta_I \theta_K}^+ P \rightarrow_{\beta_K} \\
(\text{Conservation for } \beta_I) & \quad \text{If } M \text{ is } \beta_I\text{-N then } M \text{ is } \beta_I\text{-SN} \quad \text{Barendregt’s book} \\
(\text{Conservation for } \beta + \theta) & \quad \text{If } M \text{ is } \beta_I\theta_K\text{-N then } M \text{ is } \beta\text{-SN} \quad [\text{de Groote, 1993}] \\
\end{align*}
\]

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

• $\beta_e$ is the first single relation to satisfy $\beta_K$-postponement and conservation.

• [Kamareddine, 2000] shows that:

\[
\begin{align*}
(\beta_eK\text{-postponement for } \beta_e) & \quad \text{If } M \rightarrow_{\beta_eK} N \rightarrow_{\beta_eI} O \text{ then } \exists P \text{ such that } M \rightarrow_{\beta_eI} P \rightarrow_{\beta_e}^+ \\
(\text{Conservation for } \beta_e) & \quad \text{If } M \text{ is } \beta_eI\text{-N then } M \text{ is } \beta_e\text{-SN} \\
\end{align*}
\]
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

\[
\frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash P \land Q, \Delta}
\]

- They write

\[
\frac{\Gamma \vdash_c P, \Delta \quad \Gamma \vdash_c Q, \Delta}{\Gamma \vdash_c R, \Delta}
\] if $R$ and $P \land Q$ are congruent ($R =_c P \land Q$)
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 $T$ such that
  $T, \Gamma \vdash \Delta$ 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 $\Lambda_s$, of the $\lambda$-calculus is given as follows:

  $$\Lambda_s ::= \text{IN} \mid \Lambda_s \Lambda_s \mid \lambda \Lambda_s \mid \Lambda_s \sigma^i \Lambda_s \mid \varphi_k \Lambda_s \quad \text{where } i \geq 1, \ k \geq 0.$$ 

The set of open terms, noted $\Lambda_{s_{op}}$ is given as follows:

$$\Lambda_{s_{op}} ::= \text{V} \mid \text{IN} \mid \Lambda_{s_{op}} \Lambda_{s_{op}} \mid \lambda \Lambda_{s_{op}} \mid \Lambda_{s_{op}} \sigma \Lambda_{s_{op}} \mid \varphi_k^{i} \Lambda_{s_{op}} \quad \text{where } i \geq 1, \ k \geq 0$$
$s_e$-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:

- $a \in V \cup \mathbb{N}$, i.e. $a$ is a variable or a de Bruijn number.
- $a = bc$, 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 $\bar{s}$ is a sequence of $\sigma$- and $\varphi$-items such that every pair of adjacent items in $\bar{s}$ are of the form:
  
  \[
  (\varphi^i_k)(\varphi^j_l) \text{ and } k < l \\
  (b \sigma^i)(c \sigma^j) \text{ and } i < j \\
  (\varphi^i_k)(b \sigma^j) \text{ and } k < j - 1 \\
  (b \sigma^j)(\varphi^i_k) \text{ and } j \leq k.
  \]

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

  \[
  NF ::= V | IN | (NF \delta)NF | (\lambda)NF | \bar{s}V
  \]

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

Canonical Forms [Kamareddine et al., 2001b]

- $[\ ]$ bachelors $\rightarrow$ $()[]$-pairs $(A_1)[y_1] \cdots (A_m)[y_m]$ $(B_1) \cdots (B_p)$ $x$

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

<table>
<thead>
<tr>
<th>$\theta(M)$:</th>
<th>bach. []s</th>
<th>([]) -pairs mixed with bach. []s</th>
<th>bach. ()s</th>
<th>end var</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$[x][y]$</td>
<td>$(a)[z][x'][d][y'][c][z']$</td>
<td>$(b)(e)$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\gamma(M)$:</th>
<th>bach. []s</th>
<th>([]) -pairs mixed with bach. ()s</th>
<th>bach. ()s</th>
<th>end var</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$[x][y][x']$</td>
<td>$(a)<a href="b">z</a>(c)[z'][d][y']$</td>
<td>$(e)$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\theta(\gamma(M))$:</th>
<th>bach. []s</th>
<th>([]) -pairs</th>
<th>bach. ()s</th>
<th>end var</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$[x][y][x']$</td>
<td>$(a)<a href="c">z</a>[z'][d][y']$</td>
<td>$(b)(e)$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\gamma(\theta(M))$:</th>
<th>bach. []s</th>
<th>([]) -pairs</th>
<th>bach. ()s</th>
<th>end var</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$[x][y][x']$</td>
<td>$(a)<a href="d">z</a>[y'][c][z']$</td>
<td>$(b)(e)$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

$\to_{\theta}$ and $\to_{\gamma}$ are SN et CR. So $\theta$-nf and $\gamma$-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:

$$(A_1)[y_1](A_2)[y_2]B \to_{p} (A_2)[y_2](A_1)[y_1]B \quad \text{if } y_1 \notin \text{FV}(A_2)$$
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 $B \approx_{\text{equi}} A$.

- $\rightarrow_{\theta}, \rightarrow_{\gamma}, =_{\gamma}, =_{\theta}, =_p \subset \approx_{\text{equi}} \subset =_{\beta}$ (strict inclusions).

- $A \leadsto_{\beta} B$ iff $\exists A' \in [A]. \exists B' \in [B]. A' \rightarrow_{\beta} B'$ (with compatibility)

- If $A \leadsto_{\beta} B$ then $\forall A' \in [A]. \forall B' \in [B]. A' \leadsto_{\beta} B'$.

- $\rightarrow_{\beta} \subset \rightarrow_{g} \subset \leadsto_{\beta} \subset =_{\beta} = \approx_{\beta}$.

- $\leadsto_{\beta}$ is CR: Si $A \leadsto_{\beta} B$ and $A \leadsto_{\beta} C$, then $\exists D: B \leadsto_{\beta} D$ and $C \leadsto_{\beta} D$.

- Let $r \in \{\rightarrow_{\beta}, \leadsto_{\beta}\}$. If $A \in SN_r$ and $A' \in [A]$ then $A' \in SN_r$.

- $A \in SN_{\leadsto_{\beta}}$ iff $A \in SN_{\rightarrow_{\beta}}$. 
Properties of reduction modulo classes

• $\sim_\beta$ generalises $\rightarrow_g$ and $\rightarrow_\beta$: $\rightarrow_\beta \subset \rightarrow_g \subset \sim_\beta \subset =_\beta$.

• $\approx_\beta$ and $=_\beta$ are equivalent: $A \approx_\beta B$ iff $A =_\beta B$.

• $\leadsto_\beta$ is Church Rosser:
  If $A \leadsto_\beta B$ and $A \leadsto_\beta C$, then for some $D$: $B \leadsto_\beta D$ and $C \leadsto_\beta D$.

• Classes preserve $SN_\rightarrow_\beta$: If $A \in SN_\rightarrow_\beta$ and $A' \in [A]$ then $A' \in SN_\rightarrow_\beta$.

• Classes preserve $SN_\leadsto_\beta$: If $A \in SN_\leadsto_\beta$ and $A' \in [A]$ then $A' \in SN_\leadsto_\beta$.

• $SN_\rightarrow_\beta$ and $SN_\leadsto_\beta$ are equivalent: $A \in SN_\leadsto_\beta$ iff $A \in SN_\rightarrow_\beta$.

• 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, 1910¹, 1927²] 1910–1912.

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

- Church’s **simply typed λ-calculus** $\lambda \rightarrow 1940 = \lambda$-calculus + STT.
• The hierarchies of types/orders of RTT and STT are unsatisfactory.

• The notion of function adopted in the $\lambda$-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 $\neq$ $\lambda$-calculus functions (1932).

• Not all functions need to be fully abstracted as in the $\lambda$-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.
\(\lambda\)-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 $\mathbb{N} \to \mathbb{N}$.

- $(\lambda x : \alpha . A) B \to_\beta A[x := B]$  \hspace{1cm} $(\Pi x : \alpha . A) B \to_\Pi A[x := B]$

- Write $A \to A$ as $\Pi y : A. A$ when $y$ not free in $A$. 
**The Barendregt Cube**

- **Syntax:** \( A ::= x \mid \ast \mid \Box \mid AB \mid \lambda x: A. B \mid \Pi x: A. B \)**

- **Formation rule:**
  \[
  \Gamma \vdash A : s_1 \quad \Gamma, x: A \vdash B : s_2 \quad \text{if } (s_1, s_2) \in R
  \]
  \[
  \Gamma \vdash \Pi x: A. B : s_2
  \]

<table>
<thead>
<tr>
<th>( \lambda \rightarrow )</th>
<th>Simple</th>
<th>Polymorphic</th>
<th>Dependent</th>
<th>Constructors</th>
<th>Related system</th>
<th>Refs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>((\ast, \ast))</td>
<td>((\Box, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\lambda^{\tau})</td>
<td>[Church, 1940; B]</td>
</tr>
<tr>
<td>( \lambda 2 )</td>
<td>((\ast, \ast))</td>
<td>((\Box, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(F)</td>
<td>[Girard, 1972; Re]</td>
</tr>
<tr>
<td>( \lambda P )</td>
<td>((\ast, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>(\lambda P)</td>
<td>[Bruijn, 1968; Han]</td>
</tr>
<tr>
<td>( \lambda \omega )</td>
<td>((\ast, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>[Renardel de Lavalette, Longo and Moggi]</td>
</tr>
<tr>
<td>( \lambda P_2 )</td>
<td>((\ast, \ast))</td>
<td>((\Box, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>[Coquand and Huet]</td>
</tr>
<tr>
<td>( \lambda \omega )</td>
<td>((\ast, \ast))</td>
<td>((\Box, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>[Girard, 1972]</td>
</tr>
<tr>
<td>( \lambda P_\omega )</td>
<td>((\ast, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>[Coquand and Huet]</td>
</tr>
<tr>
<td>( \lambda C )</td>
<td>((\ast, \ast))</td>
<td>((\ast, \Box))</td>
<td>((\Box, \Box))</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>(\text{AUT-QE, LF POLYREC})</td>
<td>[Coquand and Huet]</td>
</tr>
</tbody>
</table>
The Barendregt Cube

\[
\begin{align*}
\lambda \omega & \rightarrow \lambda C \\
\lambda 2 & \rightarrow \lambda P2 \\
\lambda P & \rightarrow \lambda P\omega \\
\end{align*}
\]

\[(\square, \ast) \in R \]

\[(\square, \square) \in R \]

\[(\ast, \square) \in R \]
Typing Polymorphic identity needs \((\Box, \ast)\)

\[
\begin{align*}
\text{• } & y : \ast \vdash y : \ast \quad y : \ast, x : y \vdash y : \ast \\
& \quad \quad y : \ast \vdash \Pi x : y. y : \ast \\
\text{• } & y : \ast, x : y \vdash x : y \quad y : \ast \vdash \Pi x : y. y : \ast \\
& \quad \quad y : \ast \vdash \lambda x : y. x : \Pi x : y. y \\
\vdash & * : \Box \quad y : \ast \vdash \Pi x : y. y : \ast \\
\text{• } & \vdash \Pi y : *. \Pi x : y. y : \ast \\
\text{• } & y : \ast \vdash \lambda x : y. x : \Pi x : y. y \quad \vdash \Pi y : *. \Pi x : y. y : \ast \\
& \quad \quad \vdash \lambda y : *. \lambda x : y. x : \Pi y : *. \Pi x : y. y
\end{align*}
\]

by \((\Pi)\) \((\ast, \ast)\)

by \((\lambda)\)

by \((\Pi)\) \((\Box, \ast)\)

by \((\lambda)\)
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 \(\text{let val id = (fn } x \Rightarrow x) \text{ in (id id) end}\) as this Cube term
\[
(\lambda \text{id}:(\Pi \alpha:* . \alpha \rightarrow \alpha) . \text{id}(\beta \rightarrow \beta)(\text{id } \beta))(\lambda \alpha:* . \lambda x:\alpha . x)
\]

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

ML’s typing rules forbid this expression:
\(\text{let val id = (fn } x \Rightarrow x) \text{ in (fn } y \Rightarrow y\ y)(\text{id id) end}\)
Its equivalent Cube term is this well-formed typable term of \(\lambda 2\):
\[
(\lambda \text{id} : (\Pi \alpha:* . \alpha \rightarrow \alpha).
(\lambda y : (\Pi \alpha:* . \alpha \rightarrow \alpha) . y(\beta \rightarrow \beta)(y \beta))
(\lambda \alpha:* . \text{id}(\alpha \rightarrow \alpha)(\text{id } \alpha)))
(\lambda \alpha:* . \lambda x:\alpha . x)
\]

Therefore, ML should not have the full \(\Pi\)-formation rule \((\Box, \ast)\).

ML has limited access to the rule \((\Box, \ast)\).

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 \(\lambda 2 + \lambda \omega\)).
• LF [Harper et al., 1987] is often described as $\lambda P$ of the Barendregt Cube. However, Use of $\Pi$-formation rule $(\ast, \Box)$ 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 $\Pi \alpha : \text{prop.} \ast$ of the operator Prf where for a proposition $\Sigma$, Prf$(\Sigma)$ is the type of proofs of $\Sigma$.

$$
\begin{align*}
\text{prop:} \ast & \vdash \text{prop:} \ast & \text{prop:} \ast, \alpha : \text{prop} & \vdash \ast : \Box \\
\hline
\text{prop:} \ast & \vdash \Pi \alpha : \text{prop.} \ast : \Box
\end{align*}
$$

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

• This use is in fact based on a parametric constant rather than on $\Pi$-formation.

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

• [Kamareddine et al., 2001a] precisely locate LF (between $\lambda \rightarrow$ and $\lambda P$).
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 $c(b_1, \ldots, b_n)$ with $b_1, \ldots, b_n$ terms of certain types and $c \in C$.

• $b_1, \ldots, b_n$ are called the parameters of $c(b_1, \ldots, b_n)$.

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

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

• If both $(*, s_2) \in P$ and $(\Box, s_2) \in P$ then combinations of parameters allowed. For example, it is allowed that $B_1$ has type $\ast$, whilst $B_2$ has type $\Box$. 
The Cube with parametric constants

- Let \((\ast, \ast) \subseteq \mathbb{R}, P \subseteq \{(\ast, \ast), (\ast, \square), (\square, \ast), (\square, \square)\}\).

- \(\lambda R.P = \lambda R\) and the two rules \((\overrightarrow{C}\text{-weak})\) and \((\overrightarrow{C}\text{-app})\):

\[
\Gamma \vdash b : B \quad \Gamma, \Delta_i \vdash B_i : s_i \quad \Gamma, \Delta \vdash A : s \quad (s_i, s) \in P, c\text{ is } \Gamma\text{-fresh}
\]

\[
\frac{\Gamma \vdash b : B \quad \Gamma, \Delta_i \vdash B_i : s_i \quad \Gamma, \Delta \vdash A : s}{\Gamma, c(\Delta) : A \vdash b : B}
\]

\[
\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash b_i : B_i[x_j := b_j]_j^{i-1} \quad (i = 1, \ldots, n)
\]

\[
\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash A : s \quad (\text{if } n = 0)
\]

\[
\frac{\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash c(b_1, \ldots, b_n) : A[x_j := b_j]_j^n}{\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash c(b_1, \ldots, b_n) : A[x_j := b_j]_j^n}
\]

\[\Delta \equiv x_1 : B_1, \ldots, x_n : B_n.\]

\[\Delta_i \equiv x_1 : B_1, \ldots, x_{i-1} : B_{i-1}\]
Properties of the Refined Cube

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

• (Subject Reduction SR) If $\Gamma \vdash A : B$ and $A \rightarrow^\beta A'$ then $\Gamma \vdash A' : B$

• (Strong Normalisation) For all $\vdash$-legal terms $M$, we have $\text{SN} \rightarrow^\beta (M)$.

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

• $\lambda R P$ is the system which has $\Pi$-formation rules $R$ and parameter rules $P$.

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

- \( R = \{(*, *), (*, \Box)\} \)

\[ P_1 = \emptyset \quad P_2 = \{(*, *)\} \quad P_3 = \{(*, \Box)\} \quad P_4 = \{(*, *), (*, \Box)\} \]

All \( \lambda RP_i \) for \( 1 \leq i \leq 4 \) with the above specifications are all equal in power.

- \( R_5 = \{(*, *)\} \quad P_5 = \{(*, *), (*, \Box)\} \).

\( \lambda \rightarrow < \lambda R_5 P_5 < \lambda P \): we can talk about predicates:

\[
\begin{align*}
\alpha &: * , \\
\text{eq}(x: \alpha, y: \alpha) &: * , \\
\text{refl}(x: \alpha) &: \text{eq}(x, x), \\
\text{symm}(x: \alpha, y: \alpha, p: \text{eq}(x, y)) &: \text{eq}(y, x), \\
\text{trans}(x: \alpha, y: \alpha, z: \alpha, p: \text{eq}(x, y), q: \text{eq}(y, z)) &: \text{eq}(x, z)
\end{align*}
\]

\text{eq} not possible in \( \lambda \rightarrow \).
The refined Barendregt Cube

\[(\Box, \ast) \in R\]
\[(\Box, \ast) \in P\]
\[(\ast, \Box) \in P\]
\[(\ast, \Box) \in R\]
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 $\lambda$ and $\Pi$ (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 $\lambda$- versus $\Pi$-abstraction.

- We unify the two abstractions into one.
  \[ T_b ::= \mathcal{V} \mid S \mid T_b T_b \mid b\mathcal{V}:T_b . T_b \]

- $\mathcal{V}$ is a set of variables and $S = \{\ast, \square\}$.

- The $\beta$-reduction rule becomes $\beta\quad (b_{x:A.B}C \rightarrow_b B[x := C]).$

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

- The type formation rule becomes

\[
(b_1) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (b_{x:A.B}) : s_2} \quad (s_1, s_2) \in \mathcal{R}
\]
(axiom) \[ \langle \rangle \vdash * : □ \]

(start) \[
\frac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A} \quad x \notin \text{DOM}(\Gamma)
\]

(weak) \[
\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x:C \vdash A : B} \quad x \notin \text{DOM}(\Gamma)
\]

(♭₂) \[
\frac{\Gamma, x:A \vdash b : B \quad \Gamma \vdash (♭x:A.B) : s}{\Gamma \vdash (♭x:A.b) : (♭x:A.B)}
\]

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

(conv) \[
\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}
\]
Translations between the systems with 2 binders and those with one binder

• For $A \in \mathcal{T}$, we define $\overline{A} \in \mathcal{T}_b$ as follows:
  - $\overline{s} \equiv s$  $\overline{x} \equiv x$  $\overline{AB} \equiv \overline{A} \overline{B}$
  - $\overline{\lambda x:A.B} \equiv \overline{\Pi x:A.B} \equiv \overline{b}_{\overline{x:A}}\overline{B}$.

• For contexts we define: $\langle \rangle \equiv \langle \rangle$  $\overline{\Gamma}, x : A \equiv \overline{\Gamma}, x : \overline{A}$.

• For $A \in \mathcal{T}_b$, we define $[A]$ to be $\{A' \in \mathcal{T} \text{ such that } \overline{A'} \equiv A\}$.

• For context, obviously: $[\Gamma] \equiv \{\Gamma' \text{ such that } \overline{\Gamma'} \equiv \Gamma\}$. 
Isomorphism of the cube and the $♭$-cube

- If $\Gamma \vdash A : B$ then $\Gamma \vdash♭ A : B$.

- If $\Gamma \vdash♭ A : B$ then there are unique $\Gamma' \in [\Gamma]$, $A' \in [A]$ and $B' \in [B]$ such that $\Gamma' \vdash π A' : B'$.

- The $♭$-cube enjoys all the properties of the cube except the unicity of types.
Organised multiplicity of Types for $\vdash_b$ and $\rightarrow_b$ [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 $\Gamma \vdash_b A : B_1$ and $\Gamma \vdash_b A : B_2$, then $B_1 \neq_b B_2$.

2. If $\Gamma \vdash_b A_1 : B_1$ and $\Gamma \vdash_b A_2 : B_2$ and $A_1 \equiv_b A_2$, then $B_1 \neq_b B_2$.

3. If $\Gamma \vdash_b B_1 : s_1$, $B_1 \equiv_b B_2$ and $\Gamma \vdash_b A : B_2$ then $\Gamma \vdash_b B_2 : s_1$.

4. Assume $\Gamma \vdash_b A : B_1$ and $(\Gamma \vdash_b A : B_1)^{-1} = (\Gamma', A', B_1')$. Then $B_1 \equiv_b B_2$ if:
   (a) either $\Gamma \vdash_b A : B_2$, $(\Gamma \vdash_b A : B_2)^{-1} = (\Gamma', A'', B_2')$ and $B_1' \equiv_b B_2'$,
   (b) or $\Gamma \vdash_b C : B_2$, $(\Gamma \vdash_b C : B_2)^{-1} = (\Gamma', C', B_2')$ and $A' \equiv_b C'$.
Theme 4: Extending the cube with $\Pi$-reduction loses subject reduction [Kamareddine et al., 1999b]

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

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

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

[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 ($\Pi$-reduction).
Adding type instantiation to the typing rules of the $b$-cube

If we change $(\text{app}^b)$ by (new $\text{app}^b$) in the $b$-cube we lose subject reduction.

\[(\text{app}^b) \quad \frac{\Gamma \vdash_a F : (\Pi x : A. B) \quad \Gamma \vdash_a a : A}{\Gamma \vdash_a F a : B[x := a]} \]

\[(\text{app}^{bb}) \quad \frac{\Gamma \vdash_b F : (\downarrow x : A. B) \quad \Gamma \vdash_b a : A}{\Gamma \vdash_b F a : (\downarrow x : A. B)a} \]
Failure of correctness of types and subject reduction

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

- For example, $z : \ast, x : z \vdash (\check{y} : z \cdot y)x : (\check{y} : z \cdot z)x$ yet $(\check{y} : z \cdot z)x \not\equiv \square$ and $\forall s. z : \ast, x : z \not\vdash (\check{y} : z \cdot z)x : s$.

- **Subject Reduction no longer holds.** That is, with $(\text{appl} \, \bot)$: $\Gamma \vdash A : B$ and $A \rightarrow A'$ may not imply $\Gamma \vdash A' : B$.

- For example, $z : \ast, x : z \vdash (\check{y} : z \cdot y)x : (\check{y} : z \cdot z)x$ and $(\check{y} : z \cdot y)x \rightarrow_b x$, but one can’t show $z : \ast, x : z \vdash x : (\check{y} : z \cdot z)x$. 
Solving the problem

Keep all the typing rules of the $b$-cube the same except: replace (conv) by (new-conv), (appl$^b$) by (appl$^{bb}$) and add three new rules as follows:

(start-def) \[
\begin{align*}
\Gamma \vdash A : s & \quad \Gamma \vdash B : A \\
\Gamma, x = B : A & \vdash x : A \\
x \notin \text{dom} \ (\Gamma)
\end{align*}
\]

(weak-def) \[
\begin{align*}
\Gamma \vdash A : B & \quad \Gamma \vdash C : s & \quad \Gamma \vdash D : C \\
\Gamma, x = D : C & \vdash A : B \\
\Gamma, x = B : A & \vdash C : D \\
x \notin \text{dom} \ (\Gamma)
\end{align*}
\]

(def) \[
\begin{align*}
\Gamma \vdash (\flat x : A.C)B : D[x := B]
\end{align*}
\]

(new-conv) \[
\begin{align*}
\Gamma \vdash A : B & \quad \Gamma \vdash B' : s & \quad \Gamma \vdash B =_{\text{def}} B' \\
\Gamma \vdash F : \flat x : A.B & \quad \Gamma \vdash a : A \\
\Gamma \vdash Fa : (\flat x : A.B)a
\end{align*}
\]

\textit{(appl$^{bb}$)}
In the conversion rule, \( \Gamma \vdash B =_{def} B' \) is defined as:

- If \( B =_b B' \) 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 \( \lambda \)-calculus.

- In fact, our new framework is closer to Frege's abstraction principle and the principles \( *9\cdot14 \) and \( *9\cdot15 \) of [Whitehead and Russell, 1910\(^1\), 1927\(^2\)].
Correctness of types holds.

• We demonstrate this with the earlier example.

• Recall that we have $z : *, x : z \vdash (b_{y:z}.y)x : (b_{y:z}.)x$ and want that for some $s$, $z : *, x : z \vdash (b_{y:z}.z)x : s$.

• Here is how the latter formula now holds:

$$
z : *, x : z \vdash z : * \quad \text{(start and weakening)}
$$

$$
z : *, x : z.y = x : z \vdash z : * \quad \text{(weakening)}
$$

$$
z : *, x : z \vdash (b_{y:z}.)x : *[y := x] \equiv * \quad \text{(def rule)}
$$
Subject Reduction holds.

- We demonstrate this with the earlier example.

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

- Here is how the latter formula now holds:

  a. \( z : *, x : z \vdash x : z \) (start and weakening)

  b. \( z : *, x : z \vdash (\beta_{y : z} z) x : * \) (from 1 above)

  \( z : *, x : z \vdash x : (\beta_{y : z} z) x \) (conversion, a, b, and \( z =_{\beta} (\beta_{y : z} z) x \))
He liked to join in everything
He was a great listener
He was an impressive company
He was a gentleman


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.