Types and Functions since Principia and the Computerisation of Language and Mathematics

Fairouz Kamareddine
Heriot-Watt University
Edinburgh, Scotland


23 October 2013
Summary

- General definition of function 1879 [17] is key to Frege’s formalisation of logic.

- Self-application of functions was at the heart of Russell’s paradox 1902 [51].

- To avoid paradox Russell controlled function application via type theory.

- Russell [52] 1903 gives the first type theory: the Ramified Type Theory (RTT).

- RTT is used in Russell and Whitehead’s Principia Mathematica [55] 1910–1912.


- Church’s simply typed λ-calculus $\lambda \rightarrow [12] 1940 = \lambda$-calculus + STT.
• The hierarchies of types (and orders) as found in RTT and STT are unsatisfactory.

• The notion of function adopted in the λ-calculus is unsatisfactory (cf. [30]).

• Hence, birth of different systems of functions and types, each with different functional power.

• We discuss the evolution of functions and types and their use in logic, language and computation.

• We then concentrate on these notions in mathematical vernaculars (as in Automath) and in logic.

• Frege’s functions ≠ Principia’s functions ≠ 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.

- Furthermore, 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 formal hierarchy of types.

- Another issue that we touch on is the lessons learned from formalising mathematics in logic (à la Principia) and in proof checkers (à la Automath, or any modern proof checker).
Prehistory of Types (formal systems in 19th century)

In the 19th century, the need for a more precise style in mathematics arose, because controversial results had appeared in analysis.

- 1821: Many of these controversies were solved by the work of Cauchy. E.g., he introduced a precise definition of convergence in his *Cours d’Analyse* [11].

- 1872: Due to the more exact definition of real numbers given by Dedekind [16], the rules for reasoning with real numbers became even more precise.

- 1895-1897: Cantor began formalizing set theory [9, 10] and made contributions to number theory.

- 1889: Peano formalized arithmetic [46], but did not treat logic or quantification.
Prehistory of Types (formal systems in 19th century)

• 1879: Frege was not satisfied with the use of natural language in mathematics:

  “. . . I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain the precision that my purpose required.”

  (Begriffsschrift, Preface)

Frege therefore presented *Begriffsschrift* [17], the first formalisation of logic giving logical concepts via symbols rather than natural language.

  “[Begriffsschrift's] first purpose is to provide us with the most reliable test of the validity of a chain of inferences and to point out every presupposition that tries to sneak in unnoticed, so that its origin can be investigated.”

  (Begriffsschrift, Preface)
Prehistory of Types (Begriffsschrift’s functions)

The introduction of a very general definition of function was the key to the formalisation of logic. Frege defined what we will call the Abstraction Principle.

Abstraction Principle 1.

“If in an expression, […] a simple or a compound sign has one or more occurrences and if we regard that sign as replaceable in all or some of these occurrences by something else (but everywhere by the same thing), then we call the part that remains invariant in the expression a function, and the replaceable part the argument of the function.”

(Begriffsschrift, Section 9)
Prehistory of Types (Begriffsschrift’s functions)

- Frege put *no restrictions* on what could play the role of *an argument*.
- An argument could be a *number* (as was the situation in analysis), but also a *proposition*, or a *function*.
- The *result of applying* a function to an argument did not have to be a number.
- Frege was aware of some typing rule that does not allow to substitute functions for object variables or objects for function variables:
  “Now just as functions are fundamentally different from objects, so also functions whose arguments are and must be functions are fundamentally different from functions whose arguments are objects and cannot be anything else. I call the latter *first-level*, the former *second-level.*”
  (*Function and Concept*, pp. 26–27)
Prehistory of Types (Grundgesetze’s functions)

The *Begriffsschrift*, however, was only a prelude to Frege’s writings.

- In *Grundlagen der Arithmetik* [18] he argued that mathematics can be seen as a branch of logic.
- In *Grundgesetze der Arithmetik* [19, 21] he described the elementary parts of arithmetics within an extension of the logical framework of *Begriffsschrift*.
- Frege approached the *paradox threats for a second time* at the end of Section 2 of his *Grundgesetze*.
- He did not want to *apply a function to itself*, but to its course-of-values.
- “the function $\Phi(x)$ has the same *course-of-values* as the function $\Psi(x)$” if:

  “$\Phi(x)$ and $\Psi(x)$ always have the same value for the same argument.” (Grundgesetze, p. 7)

- E.g., let $\Phi(x)$ be $x \land \neg x$, and $\Psi(x)$ be $x \leftrightarrow \neg x$, for all propositions $x$. 

PAS2013 8
Prehistory of Types (Grundgesetze’s functions)

• All essential information of a function is contained in its graph.

• So a system in which a function can be applied to its own graph should have similar possibilities as a system in which a function can be applied to itself.

• Frege excluded the paradox threats by forbidding self-application, but due to his treatment of courses-of-values these threats were able to enter his system through a back door.

• In 1902, Russell wrote to Frege [51] that he had discovered a paradox in his Begriffsschrift (Begriffsschrift does not suffer from a paradox).

• Only six days later, Frege answered that Russell’s derivation of the paradox was incorrect [20]. That self-application \( f(f) \) is not possible in the Begriffsschrift. And that Russell’s argument could be amended to a paradox in the system of his Grundgesetze, using the course-of-values of functions.
Prehistory of Types (paradox in Peano and Cantor’s systems)

• Frege’s system was not the only paradoxical one.
• The Russell Paradox can be derived in Peano’s system as well, as well as on Cantor’s Set Theory by defining the class $K \overset{\text{def}}{=} \{x \mid x \not\in x\}$ and deriving $K \in K \iff K \not\in K$.
• Paradoxes were already widely known in antiquity.
• The oldest logical paradox: the Liar’s Paradox “This sentence is not true”, also known as the Paradox of Epimenides. It is referred to in the Bible (Titus 1:12) and is based on the confusion between language and meta-language.
• The Burali-Forti paradox ([8], 1897) is the first of the modern paradoxes. It is a paradox within Cantor’s theory on ordinal numbers.
• Cantor’s paradox on the largest cardinal number occurs in the same field. It discovered by Cantor around 1895, but was not published before 1932.
Prehistory of Types (paradoxes)

• Logicians considered these paradoxes to be out of the scope of logic: The Liar’s Paradox can be regarded as a problem of linguistics. The paradoxes of Cantor and Burali-Forti occurred in what was considered in those days a highly questionable part of mathematics: Cantor’s Set Theory.

• The Russell Paradox, however, was a paradox that could be formulated in all the systems that were presented at the end of the 19th century (except for Frege’s Begriffsschrift). It was at the very basics of logic. It could not be disregarded, and a solution to it had to be found.

• In 1903-1908, Russell suggested the use of types to solve the problem [53].
Prehistory of Types (vicious circle principle)

“In all the above contradictions there is a common characteristic, which we may describe as self-reference or reflexiveness. [...] In each contradiction something is said about all cases of some kind, and from what is said a new case seems to be generated, which both is and is not of the same kind as the cases of which all were concerned in what was said.”

(Mathematical logic as based on the theory of types)

Russell’s plan was, to avoid the paradoxes by avoiding all possible self-references. He postulated the “vicious circle principle”:

“Whatever involves all of a collection must not be one of the collection.”

(Mathematical logic as based on the theory of types)

Russell implements this principle very strictly using types.
Problems of Ramified Type Theory

- The main part of the *Principia* is devoted to the development of logic and mathematics using the legal pfs of the ramified type theory.

- *ramification*/division of simple types into orders make RTT not easy to use.

- **(Equality)** \( x \equiv_L y \overset{\text{def}}{\leftrightarrow} \forall z [z(x) \leftrightarrow z(y)] \).
  
  In order to express this general notion in RTT, we have to incorporate all pfs \( \forall z : (0^0)^n [z(x) \leftrightarrow z(y)] \) for \( n > 1 \), and this cannot be expressed in one pf.

- Not possible to give a constructive proof of the theorem of the least upper bound within a ramified type theory.

- It is not possible in RTT to give a definition of an object that refers to the class to which this object belongs (because of the Vicious Circle Principle). Such a definition is called an *impredicative definition*. 
Axiom of Reducibility

• Russell and Whitehead tried to solve problems with the axiom of reducibility: *For each formula \( f \), there is a formula \( g \) with a predicative type such that \( f \) and \( g \) are (logically) equivalent.*

• The validity of the Axiom of Reducibility has been questioned from the moment it was introduced.

• Though Weyl [54] made an effort to develop analysis within the Ramified Theory of Types (without the Axiom of Reducibility),

• and various parts of mathematics can be developed within \( \text{RTT} \) and without the Axiom,

• the general attitude towards \( \text{RTT} \) (without the axiom) was that the system was too restrictive, and that a *better solution* had to be found.
Deramification

- Ramsey considers it essential to divide the paradoxes into two parts:

- **logical or syntactical** paradoxes (like the Russell paradox, and the Burali-Forti paradox) are removed

  "by pointing out that a propositional function cannot significantly take itself as argument, and by dividing functions and classes into a hierarchy of types according to their possible arguments."

  (The Foundations of Mathematics, p. 356)

- **Semantical** paradoxes are excluded by the hierarchy of orders. These paradoxes (like the Liar’s paradox, and the Richard Paradox) are based on the confusion of language and meta-language. These paradoxes are, therefore, not of a purely mathematical or logical nature. When a proper distinction between object language and meta-language is made, these so-called semantical paradoxes disappear immediately.
The Simple Theory of Types

- Ramsey [47], and Hilbert and Ackermann [26], simplified the Ramified Theory of Types RTT by removing the orders. The result is known as the Simple Theory of Types (STT).

- Nowadays, STT is known via Church’s formalisation in $\lambda$-calculus. However, STT already existed (1926) before $\lambda$-calculus did (1932), and is therefore not inextricably bound up with $\lambda$-calculus.

- How to obtain STT from RTT? Just leave out all the orders and the references to orders (including the notions of predicative and impredicative types).
Limitation of the simply typed $\lambda$-calculus

- $\lambda \rightarrow$ is very restrictive.
- Numbers, booleans, the identity function have to be defined at every level.
- We can represent (and type) terms like $\lambda x : o.x$ and $\lambda x : \iota.x$.
- We cannot type $\lambda x : \alpha.x$, where $\alpha$ can be instantiated to any type.
- This led to new (modern) type theories that allow more general notions of functions (e.g, polymorphic).
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.
Functionalisation and Instantiation

[35] assessed evolution of the function concept from two points of view:

• **Functionalisation**: the construction of a function out of an expression, as in constructing the function $\lambda_x.x \times 3 + x$ from the expression $2 \times 3 + 2$.

• Functionalisation is
  – *Abstraction from a subexpression* e.g., moving from $2 \times 3 + 2$ to $x \times 3 + x$
  – *Function construction* e.g., turning $x \times 3 + x$ into $\lambda_x.x \times 3 + x$.

• **Instantiation**: the calculation of a function value when a suitable argument is assigned to the function, as in the construction of $2 \times 3 + 2$ by applying the function $\lambda_x.x \times 3 + x$ to 2.

• Instantiation is:
  – *Application construction* e.g., $(\lambda_x.x \times 3 + x)2$ the application of $\lambda_x.x \times 3 + x$ to 2
  – *Concretisation to a subexpression* e.g., calculating $(\lambda_x.x \times 3 + x)2$ to $2 \times 3 + 2$. 
\(\lambda\)-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 $A : \ast$ for $A$ is a type)
  
  – $\lambda_y:A.y$, the identity over $A$ has type $A \rightarrow A$
  
  – $\lambda A:\ast.\lambda y:A.y$, the polymorphic identity has type $\Pi A:\ast.A \rightarrow A$

• 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 $(A \rightarrow A)[A := \mathbb{N}]$ or $\mathbb{N} \rightarrow \mathbb{N}$.
  
  – $(\lambda A:\ast.\lambda y:A.y)\mathbb{N}$ has type $(\Pi A:\ast.A \rightarrow A)\mathbb{N} = (A \rightarrow A)[A := \mathbb{N}]$ or $\mathbb{N} \rightarrow \mathbb{N}$.

• $(\lambda x:\alpha.A)B \rightarrow_\beta A[x := B]$  
  $(\Pi x:\alpha.A)B \rightarrow_{\Pi} A[x := B]$

• Write $A \rightarrow A$ as $\Pi y:A.A$ when $y$ not free in $A$. 
The Barendregt Cube

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

- 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>Constructor</th>
<th>Simple</th>
<th>Poly-morphic</th>
<th>Dependent</th>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda \rightarrow )</td>
<td>(( \ast, \ast ))</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
</tr>
<tr>
<td>( \lambda_2 )</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
</tr>
<tr>
<td>( \lambda P )</td>
<td>(( \ast, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
<td>( \text{AUT-QE, LF} )</td>
</tr>
<tr>
<td>( \lambda \omega )</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
</tr>
<tr>
<td>( \lambda P_2 )</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
</tr>
<tr>
<td>( \lambda \omega )</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
</tr>
<tr>
<td>( \lambda P_\omega )</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
</tr>
<tr>
<td>( \lambda C )</td>
<td>(( \ast, \ast ))</td>
<td>(( \square, \ast ))</td>
<td>(( \ast, \square ))</td>
<td>(( \square, \square ))</td>
</tr>
</tbody>
</table>
The Barendregt Cube

\[ \lambda \rightarrow \lambda \]

\[ \lambda \omega \]

\[ \lambda C \]

\[ \lambda P2 \]

\[ \lambda P \omega \]

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

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

\[ (*, \Box) \in R \]
Typing Polymorphic identity needs (□, *)

\[
\begin{align*}
\alpha & : \Box & \alpha : \Box & \vdash \alpha : \Box \\
\beta & : \Box & \beta : \Box, x : \gamma & \vdash \beta : \Box \\
\gamma & : \Box & \vdash \Pi x : \gamma . \gamma : \Box \\
\delta & : \Box & \delta : \Box & \vdash \Pi x : \delta . \delta : \Box \\
\end{align*}
\]

by (Π) (□, *)

\[
\begin{align*}
\eta & : \Box & \eta : \Box & \vdash \eta : \Box \\
\eta & : \Box & \eta : \Box, x : \gamma & \vdash \eta : \Box \\
\gamma & : \Box & \vdash \lambda x : \gamma . x : \Pi x : \gamma.y \\
\gamma & : \Box & \vdash \Lambda x : \gamma . \gamma : \Box \\
\end{align*}
\]

by (λ)

\[
\begin{align*}
\conjunction & : \Box & \conjunction : \Box & \vdash \conjunction : \Box \\
\conjunction & : \Box & \conjunction : \Box & \vdash \Pi y : \conjunction . \Pi x : \conjunction.y : \Box \\
\end{align*}
\]

by (Π) (□, *)

\[
\begin{align*}
\eta & : \Box & \eta : \Box & \vdash \eta : \Box \\
\eta & : \Box & \eta : \Box, x : \gamma & \vdash \eta : \Box \\
\gamma & : \Box & \vdash \lambda x : \gamma . x : \Pi x : \gamma.y \\
\gamma & : \Box & \vdash \Lambda y : \gamma . \lambda x : \gamma . x : \Pi y : \conjunction . \Pi x : \gamma.y \\
\end{align*}
\]

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 view of ML

- The language ML is not based on all of system F (2nd 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 ⇒ x) in (id id) end as this Cube term
  $(\lambda id : (\Pi \alpha : *. \alpha \rightarrow \alpha). \text{id}(\beta \rightarrow \beta)(\text{id} \beta))(\lambda \alpha : *. \lambda x : \alpha. x)$
• To type this in the Cube, the \((\square, \ast)\) rule is needed (i.e., \(\lambda 2\)).

• ML’s typing rules forbid this expression:
  
  \[
  \text{let val id} = (\text{fn } x \Rightarrow x) \text{ in } (\text{fn } y \Rightarrow y\ y)(\text{id } \text{id}) \text{ end}
  \]

  Its equivalent Cube term is this well-formed typable term of \(\lambda 2\):
  
  \[
  (\lambda \text{id } : (\Pi \alpha : \ast. \alpha \rightarrow \alpha).
  \quad (\lambda y : (\Pi \beta : \ast. \alpha \rightarrow \beta). y\ (\beta \rightarrow \beta)) (y\ \beta))
  \quad (\lambda \alpha : \ast. \text{id}(\alpha \rightarrow \alpha)(\text{id } \alpha)))
  \quad (\lambda \alpha : \ast. \lambda x : \alpha. x)
  \]

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

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

• ML’s type system is none of those of the eight systems of the Cube.

• [29] places the type system of ML (between \(\lambda 2 + \lambda \omega\)).
LF

• LF [24] is often described as $\lambda P$ of the Barendregt Cube. However, Use of \( \Pi \)-formation rule \((*, \Box)\) is restricted in LF [22].

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

\[
\begin{align*}
\text{prop}:* & \vdash \text{prop}:* \\
\text{prop}:* & \vdash \alpha:\text{prop} \vdash *:\Box \\
\text{prop}:* & \vdash \Pi \alpha:\text{prop}.* : \Box
\end{align*}
\]

• In LF, this is the only point where the \( \Pi \)-formation rule \((*, \Box)\) is used. But, \( \text{Prf} \) is only used when applied to \( \Sigma:\text{prop} \). We never use \( \text{Prf} \) 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 \( \text{Prf} \) in a parametric form, and use \((*, \Box)\) as a parameter instead of a \( \Pi \)-formation rule.

• [29] precisely locate LF (between $\lambda \to$ 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* [42].

- 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 [4] 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 [29]

- 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 \( c(b_1, \ldots, b_n) \).

- \( \mathcal{S} \) allows several kinds of \( \Pi \)-constructs. We also use a set \( P \) of \((s_1, s_2)\) where \( s_1, s_2 \in \{\ast, \Box\} \) 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 \((\ast, s_2) \in P\) 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 \((*,*) \subseteq \mathcal{G}, P \subseteq \{(*, *), (*, \square), (\square, *), (\square, \square)\}\).

• \(\lambda \mathcal{G}P = \lambda \mathcal{G}\) and the two rules \((\mathcal{C}\text{-weak})\) and \((\mathcal{C}\text{-app})\):

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

\[
\frac{
\Gamma, c(\Delta) : A \vdash b : B
}{
\Gamma \vdash b : B
}\]

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

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

\[
\frac{
\Gamma_1, c(\Delta)_i : A, \Gamma_2 \vdash c(b_1, \ldots, b_n) : A[x_j:=b_j]_{j=1}^{n}
}{
\Delta \equiv x_1:B_1, \ldots, x_n:B_n.
\Delta_i \equiv x_1:B_1, \ldots, x_{i-1}:B_{i-1}
}\]
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 $A \rightarrow^\beta A'$ 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.

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

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

- $R = \{(*, *), (*, □)\}$

$P_1 = \emptyset \quad P_2 = \{(*, *)\} \quad P_3 = \{(*, □)\} \quad P_4 = \{(*, *), (*, □)\}$

All $\lambda R P_i$ for $1 \leq i \leq 4$ with the above specifications are all equal in power.

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

$\lambda \rightarrow < \lambda R_5 P_5 < \lambda P$: 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 $\lambda \rightarrow$. 

PAS2013 34
The refined Barendregt Cube
LF, ML, Aut-68, and Aut-QE in the refined Cube
Logicians versus mathematicians and induction over numbers

- Logician uses \textit{ind}: \textit{Ind} as proof term for an application of the induction axiom. The type \textit{Ind} can only be described in \( \lambda R \) where \( R = \{ (*, *), (\Box, *), (\Box, \Box) \} \):
  \[
  \text{Ind} = \Pi p:(N \to \ast). p0 \to (\Pi n: N. \Pi m: N. pn \to Smn \to pm) \to \Pi n: N. pn \quad (1)
  \]
- Mathematician uses \textit{ind} only with \( P : N \to \ast \), \( Q : P0 \) and \( R : (\Pi n: N. \Pi m: N. Pn \to Smn \to Pm) \) to form a term \((\text{ind}PQR):(\Pi n: N. Pn)\).

- The use of the induction axiom by the mathematician is better described by the parametric scheme (\( p \), \( q \) and \( r \) are the \textit{parameters} of the scheme):
  \[
  \text{ind}(p: N \to \ast, q: p0, r: (\Pi n: N. \Pi m: N. pn \to Smn \to pm)) : \Pi n: N. pn \quad (2)
  \]
- The logician’s type \textit{Ind} is not needed by the mathematician and the types that occur in 2 can all be constructed in \( \lambda R \) with \( R = \{ (*, *)(*, \Box) \} \).
Logicians versus mathematicians and induction over numbers

• Mathematician: only *applies* the induction axiom and doesn’t need to know the proof-theoretical backgrounds.

• A logician develops the induction axiom (or studies its properties).

• $(\Box, \ast)$ is not needed by the mathematician. It is needed in logician’s approach in order to form the $\Pi$-abstraction $\Pi p: (\mathbb{N} \rightarrow \ast). \cdots$.

• Consequently, the type system that is used to describe the mathematician’s use of the induction axiom can be weaker than the one for the logician.

• Nevertheless, the parameter mechanism gives the mathematician limited (but for his purposes sufficient) access to the induction scheme.
• 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 [42].

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

• Parameters enable us to find an exact position of type systems in the generalised framework of type systems.

• Parameters describe the difference between developers and users of systems.
**Identifying $\lambda$ and $\Pi$ (see [33])**

- 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.

$$T_b ::= \mathcal{V} | S | T_b T_b | b \mathcal{V}:T_b.T_b$$

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

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

- 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

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

(start) \[ \frac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A} \quad x \not\in \text{dom} (\Gamma) \]

(weak) \[ \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x:C \vdash A : B} \quad x \not\in \text{dom} (\Gamma) \]

(\flat_2) \[ \frac{\Gamma, x:A \vdash b : B \quad \Gamma \vdash (\flat x:A.B) : s}{\Gamma \vdash (\flat x:A.b) : (\flat x:A.B)} \]

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

(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 $A \in T$, we define $\overline{A} \in T_b$ as follows:
  - $\overline{s} \equiv s \quad \overline{x} \equiv x \quad \overline{AB} \equiv \overline{A} \overline{B}$
  - $\overline{\lambda x:A.B} \equiv \overline{\Pi x:A.B} \equiv \overline{b}_{x:\overline{A}}.\overline{B}$.

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

- For $A \in T_b$, we define $[A]$ to be $\{A' \in T$ 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 $\Gamma \vdash_♭ \overline{A} : \overline{B}$.

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

- The $♭$-cube enjoys all the properties of the cube except the unicity of types.
Organised multiplicity of Types for $\vdash_b$ and $\rightarrow_b$ [33]

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 $B_1 \equiv_b B_2$.

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

3. If $\Gamma \vdash_b B_1 : s_1$, $B_1 =_b B_2$ 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 $B_1 =_b B_2$ if:
   
   (a) either $\Gamma \vdash_b A : B_2$, $(\Gamma \vdash_b A : B_2)^{-1} = (\Gamma', A'', B'_2)$ and $B'_1 =_\beta B'_2$,
   
   (b) or $\Gamma \vdash_b C : B_2$, $(\Gamma \vdash_b C : B_2)^{-1} = (\Gamma', C', B'_2)$ and $A' =_\beta C'$.
Extending the cube with $\Pi$-reduction loses subject reduction [34]

If we change \textit{(appl)} by \textit{(new appl)} in the cube we lose subject reduction.

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

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

[34] 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.
Adding type instantiation to the typing rules of the $b$-cube

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

$\begin{align*}
(\text{app}^b) & \quad \frac{\Gamma \vdash_b F : (\Pi x : A . B) \quad \Gamma \vdash_b a : A}{\Gamma \vdash_b Fa : B[x := a]} \\
(\text{app}^{\text{new}b}) & \quad \frac{\Gamma \vdash_b F : (b x : A . B) \quad \Gamma \vdash_b a : A}{\Gamma \vdash_b Fa : (b x : A . B)a}
\end{align*}$
Failure of correctness of types and subject reduction

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

• For example, $z : *, x : z \vdash (♭_y : z.y)x : (♭_y : z.z)x$ yet $(♭_y : z.z)x \not\equiv \Box$ and $\forall s. z : *, x : z \nvdash (♭_y : z.z)x : s$.

• Subject Reduction no longer holds. That is, with (appl♭): $\Gamma \vdash A : B$ and $A \rightarrow A'$ may not imply $\Gamma \vdash A' : B$.

• For example, $z : *, x : z \vdash (♭_y : z.y)x : (♭_y : z.z)x$ and $(♭_y : z.y)x \rightarrow♭ x$, but one can’t show $z : *, x : z \vdash x : (♭_y : z.z)x$. 

PAS2013
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) \[
\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : A}{\Gamma, x = B : A \vdash x : A} \quad x \notin \text{DOM}(\Gamma)
\]

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

(def) \[
\frac{\Gamma \vdash (b_{x:A}.C)B : D[x:=B]}{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad \Gamma \vdash B =_{def} B'}
\]

(new-conv) \[
\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad \Gamma \vdash B =_{def} B'}{\Gamma \vdash F : b_{x:A}.B \quad \Gamma \vdash a : A}
\]

(appl$^{bb}$) \[
\frac{\Gamma \vdash F : b_{x:A}.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : (b_{x:A}.B)a}
\]
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 $\ast 9.14$ and $\ast 9.15$ of [55].**
Correctness of types holds.

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

\[
\begin{align*}
  z : *, x : z & \vdash z : * & \text{(start and weakening)} \\
  z : *, x : z. y : z \langle x & \vdash z : * & \text{(weakening)} \\
  z : *, x : z & \vdash (b_{y : z} z) x : *[y := x] \equiv * & \text{(def rule)}
\end{align*}
\]
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 $z \equiv_{\beta} (\beta_{y:z} z)x$)
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:
They look good together
Theme 1: De Bruijn Indices and Explicit Substitutions [14]

- 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 $x, y, z, \ldots$).

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

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

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

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

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

- $(A_1A_2)\{i \leftarrow B\} = (A_1\{i \leftarrow B\})(A_2\{i \leftarrow B\})$
  $(\lambda A)\{i \leftarrow B\} = \lambda(A\{i + 1 \leftarrow B\})$
  $n\{i \leftarrow B\} = \begin{cases} 
  n - 1 & \text{if } n > i \\
  U^i_0(B) & \text{if } n = i \\
  n & \text{if } n < i.
\end{cases}$

- 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 $\lambda s$ [31]

- Write $A\{n \leftarrow B\}$ as $A\sigma^n B$ and $U_k^i(A)$ as $\varphi_k^i A$.
- $A ::= n | (\lambda B) | (BC') | (A\sigma^i B) | (\varphi_k^i B)$ where $i, n \geq 1$, $k \geq 0$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntax</th>
<th>Meaning</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>
<tr>
<td>$\sigma$-destruction</td>
<td>$n \sigma^i B$</td>
<td>$\begin{cases} n - 1 &amp; \text{if } n &gt; i \ \varphi_0^i B &amp; \text{if } n = i \ n &amp; \text{if } n &lt; i \end{cases}$</td>
</tr>
<tr>
<td>$\varphi$-$\lambda$-transition</td>
<td>$\varphi_k^i (\lambda A)$</td>
<td>$\lambda(\varphi_{k+1}^i A)$</td>
</tr>
<tr>
<td>$\varphi$-app-transition</td>
<td>$\varphi_k^i (A_1 A_2)$</td>
<td>$(\varphi_k^i A_1)(\varphi_k^i A_2)$</td>
</tr>
<tr>
<td>$\varphi$-destruction</td>
<td>$\varphi_k^i n$</td>
<td>$\begin{cases} n + i - 1 &amp; \text{if } n &gt; k \ n &amp; \text{if } n \leq k \end{cases}$</td>
</tr>
</tbody>
</table>
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.
$\lambda v$ [3]

Terms: $\Lambda v^t ::= IN \mid \Lambda v^t \Lambda v^t \mid \lambda \Lambda v^t \mid \Lambda v^t[\Lambda v^s]$

Substitutions: $\Lambda v^s ::= \uparrow \mid \uparrow (\Lambda v^s) \mid \Lambda v^t$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Reduces to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Beta)</td>
<td>$(\lambda a)b$</td>
<td>$a[b/]$</td>
</tr>
<tr>
<td>(App)</td>
<td>$(ab)[s]$</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>(FVar)</td>
<td>$1[a/]$</td>
<td>$a$</td>
</tr>
<tr>
<td>(RVar)</td>
<td>$n+1[a/]$</td>
<td>$n$</td>
</tr>
<tr>
<td>(FVarLift)</td>
<td>$1[\uparrow(s)]$</td>
<td>$1$</td>
</tr>
<tr>
<td>(RVarLift)</td>
<td>$n+1[\uparrow(s)]$</td>
<td>$n[s][\uparrow]$</td>
</tr>
<tr>
<td>(VarShift)</td>
<td>$n[\uparrow]$</td>
<td>$n+1$</td>
</tr>
</tbody>
</table>

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

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

\[
\begin{align*}
(Beta) & \quad (\lambda a) b & \rightarrow & \ a \ [b \cdot id] \\
(App) & \quad (a \ b)[s] & \rightarrow & \ (a \ [s]) \ (b \ [s]) \\
(Abs) & \quad (\lambda a)[s] & \rightarrow & \ \lambda(a \ [\uparrow(s)]) \\
(Clos) & \quad (a \ [s])[t] & \rightarrow & \ a \ [s \circ t] \\
(Varshift1) & \quad n[\uparrow] & \rightarrow & \ n + 1 \\
(Varshift2) & \quad n[\uparrow \circ s] & \rightarrow & \ n + 1 \ [s] \\
(FVarCons) & \quad 1[a \cdot s] & \rightarrow & \ a \\
(RVarCons) & \quad n + 1[a \cdot s] & \rightarrow & \ n \ [s] \\
(FVarLift1) & \quad 1[\uparrow(s)] & \rightarrow & \ 1 \\
(FVarLift2) & \quad 1[\uparrow(s) \circ t] & \rightarrow & \ 1 \ [t] \\
(RVarLift1) & \quad n + 1[\uparrow(s)] & \rightarrow & \ n[s \circ \uparrow] \\
(RVarLift2) & \quad n + 1[\uparrow(s) \circ t] & \rightarrow & \ n[s \circ (\uparrow \circ t)]
\end{align*}
\]
$\lambda_\sigma_\uparrow$ rules continued

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

$\lambda_\sigma_\uparrow$ satisfies 1., 2., and 4., but does not have PSN.
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 \( \lambda s \)?

- \( A ::= X \mid n \mid (\lambda B) \mid (BC) \mid (A \tau^i B) \mid (\varphi_k B) \) 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):
  \[
  ((\lambda X)Y)\sigma^1 1 \rightarrow (X\sigma^1 Y)\sigma^1 1 \quad ((\lambda X)Y)\sigma^1 1 \rightarrow ((\lambda X)\sigma^1 1)(Y\sigma^1 1)
  \]

- \((X\sigma^1 Y)\sigma^1 1\) 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 \( \lambda s \):
• These extra rules are the rewriting of the well-known meta-substitution ($\sigma - \sigma$) and distribution ($\sigma - \sigma$) lemmas (and the 4 extra lemmas needed to prove them).

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

• ($\sigma - \sigma$):
  
Where did the extra rules come from?

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

(σ − ϕ 1) For \( k < j < k + i \) we have: \( U_{k}^{i-1}(A) = U_{k}^{i}(A)\{j \leftarrow B\} \).

(ϕ − ϕ 2) For \( l \leq k < l + j \) we have: \( U_{k}^{i}(U_{l}^{j}(A)) = U_{l}^{j+i-1}(A) \).

(σ − ϕ 2) For \( k + i \leq j \) we have: \( U_{k}^{i}(A)\{j \leftarrow B\} = U_{k}^{i}(A\{j - i + 1 \leftarrow B\}) \).

(σ − σ) \textbf{[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 \((σ − σ)\) uses \((σ − ϕ 1)\) and \((σ − ϕ 2)\) both with \( k = 0 \).
- The proof of \((σ − ϕ 2)\) requires \((ϕ − ϕ 2)\) with \( l = 0 \).
Where did the extra rules come from (continued)?

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

$$ (\varphi - \varphi_1) \text{ For } j \leq k + 1 \text{ we have: } U^i_{k+p}(U^j_p(A)) = U^j_p(U^i_{k+p+1-j}(A)). $$

$$ (\varphi - \sigma) \text{ [Distribution lemma] } $$

For $j \leq k + 1$ we have: $U^i_k(A\{j \leftarrow B\}) = U^i_{k+1}(A)\{j \leftarrow U^i_{k+1-j}(B)\}.$

- $(\varphi - \varphi_1)$ with $p = 0$ is needed to prove $(\varphi - \sigma)$. 

PAS2013 65
Theme 2: Lambda Calculus à la de Bruijn

- \( I(x) = x, \quad I(\lambda x. B) = [x]I(B), \quad I(AB) = \langle I(B) \rangle I(A) \)

- \((\lambda x. \lambda y. xy) z\) 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 \(((\lambda x. (\lambda y. \lambda z. \ldots ) c) b) a\), 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
\[ \frac{((\lambda_x. (\lambda_y. \lambda_z. zd) c) b) a}{\downarrow \beta} \]
\[ \frac{((\lambda_y. \lambda_z. zd) c) a}{\downarrow \beta} \]
\[ \frac{(\lambda_z. zd) a}{\downarrow \beta} \]
\[ \frac{ad}{\downarrow \beta} \]

de Bruijn's Notation
\[ \frac{\langle a \rangle \langle b \rangle [x] \langle c \rangle [y] [z] \langle d \rangle z}{\downarrow \beta} \]
\[ \frac{\langle a \rangle \langle c \rangle [y] [z] \langle d \rangle z}{\downarrow \beta} \]
\[ \frac{\langle a \rangle [z] \langle d \rangle z}{\downarrow \beta} \]
\[ \frac{\langle d \rangle a}{\downarrow \beta} \]

- This makes it easy to introduce local/global/mini reductions into the λ-calculus [7].
- Further study of de Bruijn’s notation can be found in [36, 37]
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>(θ)</td>
<td>$((\lambda_x.N)P)Q$</td>
<td>$\langle Q\rangle\langle P\rangle[x]N$</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>(γ)</td>
<td>$(\lambda_x.\lambda_y.N)P$</td>
<td>$\langle P\rangle[x][y]N$</td>
</tr>
<tr>
<td></td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td></td>
<td>$\lambda_y.(\lambda_x.N)P$</td>
<td>$[y]\langle P\rangle[x]N$</td>
</tr>
<tr>
<td>(γC)</td>
<td>$((\lambda_x.\lambda_y.N)P)Q$</td>
<td>$\langle Q\rangle\langle P\rangle[x][y]N$</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>(g)</td>
<td>$((\lambda_x.\lambda_y.N)P)Q$</td>
<td>$\langle Q\rangle\langle P\rangle[x][y]N$</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>(βe)</td>
<td>?</td>
<td>$\langle Q\rangle\overline{s}[y]N$</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

- [48] uses $\theta$ and $\gamma$ in analyzing perpetual reduction strategies.
- Term reshuffling is used in [41, 39] in analyzing typability problems.
- [44, 15, 40, 32] use generalised reduction and/or term reshuffling in relating SN to WN.
- [1] uses a form of term-reshuffling in obtaining a calculus that corresponds to lazy functional evaluation.
- [36, 28, 38, 5] 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
Even more: de Bruijn’s generalised reduction has better properties

\[(\beta) \quad (\lambda x. M) N \to M[x := N]\]

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

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

\[(\theta) \quad (\lambda x. N) PQ \to (\lambda x. NQ) P\]

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

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

- Conservation of $\beta_e$: If $A$ is $\beta_eI$-normalisable then $A$ is $\beta_e$-strongly normalisable.

- Postponement 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 $r \in \{\beta_I, \theta_K\}$).

$(\beta_K$-postponement for $r)$ \quad If $M \to_{\beta_K} N \to_r O$ then $\exists P$ such that $M \to^{+\beta_I\theta_K}_P P \to_{\beta_K}$

(Conservation for $\beta_I$) \quad If $M$ is $\beta_I$-N then $M$ is $\beta_I$-SN \quad Barendregt’s book

(Conservation for $\beta + \theta$) \quad If $M$ is $\beta_I\theta_K$-N then $M$ is $\beta$-SN \quad [15]

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.

[32] shows that:

$(\beta_{eK}$-postponement for $\beta_e)$ \quad If $M \to_{\beta_{eK}} N \to_{\beta_eI} O$ then $\exists P$ such that $M \to_{\beta_eI} P \to^{+\beta_e}_P$

(Conservation for $\beta_e$) \quad If $M$ is $\beta_{eI}$-N then $M$ is $\beta_e$-SN
Here is 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
References


Published as Mathematical Centre Tracts nr. 83 (Amsterdam, Mathematisch Centrum, 1979).


