Themes in the λ-Calculus and Type Theory

Fairouz Kamareddine (Heriot-Watt University)

April 2008

A Refinement of Barendregt's Cube with Non-First-Class Functions (see book [Kamareddine et al.])

- *General definition of function* is key to Frege's *formalisation of logic* (1879).
- *Self-application of functions* was at the heart of *Russell's paradox* (1902).
- To avoid paradoxes, Russell controlled function application via *type theory*.
- Russell (1908) gives the first type theory: the *Ramified Type Theory* (RTT).
- RTT is used in Russell and Whitehead's *Principia Mathematica* (1910–1912).
- *Simple theory of types* (STT): Ramsey (1926), Hilbert and Ackermann (1928).
- Frege's functions $\neq$ Principia's functions $\neq$ *λ-calculus* functions (1932).

- Church's *simply typed λ-calculus* $\lambda{\to} = \lambda$-calculus + STT (1940).

- Both RTT and STT are *unsatisfactory*. Hence, birth of *different type systems*, each with *different functional power*. All based on Church's λ-calculus.

- *Eight influential typed λ-calculi* 1940–1988 unified in *Barendregt's cube*.

- 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.

- We extend the cube of the eight influential type systems with non-first-class functions showing that this allows placing the type systems of ML, LF and Automath more accurately in the hierarchy of types.

Prehistory of Types (Paradox Threats)

- *Types* have *always existed* in mathematics, but not explicited until 1879. Euclid avoided *impossible* situations (e.g., two parallel points) via classes/*types*.
- In formal systems, intuition can't use implicit types to avoid impossibilities.
- In 19th century, controversies in analysis led to mathematical *precision*. (*Cauchy*, *Dedekind*, *Cantor*, *Peano*, *Frege*).
- Frege's *general definition of function* was key to his formalisation of logic.

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."*

Prehistory of Types (Begriffsschrift/Grundgesetze)

- *An argument* could be a *number* (as in analysis), a *proposition*, or a *function*.

- Distinguishing *1st- and 2nd-level objects avoids paradox in Begriffsschrift*:

  "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*."

- In Grundgesetze Frege described arithmetic in an extension of *Begriffsschrift*.

- To avoid *paradox*, he *applied a function to its course-of-values*, not *itself*

- Frege treated *courses-of-values* as *ordinary objects*. Hence, *a function that takes objects as arguments could have its own course-of-values as an argument*.

Russell's Paradox, vicious circle principle

- In 1902, Russell wrote Frege saying he *discovered a paradox* in *Begriffsschrift* using $f(x) = \neg x(x)$, (*Begriffsschrift does not suffer from a paradox*).
- Frege replied: *Russell's derivation was incorrect*, $f(f)$ *is not possible in* Begriffsschrift: $f(x)$ needs *objects* as arguments; functions are not objects.
- Using *courses-of-values*, *Russell's argument gives a paradox in* Grundgesetze

Russell *avoided all possible self-references* by the *"vicious circle principle VCP"*:

"Whatever involves all of a collection must not be one *of the collection.*"

- VCP implemented by a double hierarchy of *types*: *(simple) types* and *orders*.
- The ideas behind simple types was already explained by Frege.
- Due to problems with RTT, the *(Axiom of Reducibility AR)* was introduced
  "For each formula f, there is a formula g with a *predicative* type such that f and g are (logically) equivalent."
- RTT without AR was considered too restrictive and AR itself was questionned.

- Ramsey distinguishes the logical/syntactical and semantical paradoxes.

- RTT without orders eliminates logical paradoxes. Separating language and meta language eliminates semantical paradoxes. No need for orders.

- Simple Theory of Types *(STT)* is RTT without orders.

- STT is not *Church's $\lambda \rightarrow$*. STT *existed (1926) before λ-calculus (1932)*.

The evolution of functions with Frege 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.

- In *Principia Mathematica* [Whitehead and Russell, 1910^1, 1927^2]: If, for some a, there is a proposition ϕa, then there is a function $\phi\hat{x}$, and vice versa.
- The function ϕ is not a separate entity but always has an argument.
- Frege denoted the course-of-values (*graph*) of a function $\Phi(x)$ by $\grave{\varepsilon}\Phi(\varepsilon)$.
- $\grave{\varepsilon}\Phi(\varepsilon)$ may have given Russell's $\hat{x}\Phi(x)$ for the class of objects with property Φ.
- According to Rosser, the notation $\hat{x}\Phi(x)$ is the basis of the notation $\lambda x.\Phi$.
- Church wrote $\wedge x\Phi(x)$ for $x \mapsto \Phi(x)$ to distinguish it from the class $\hat{x}\Phi(x)$.

λ-calculus does not fully represent functionalisation

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

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

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

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

- cannot abstract only half way: $x + 3$ is not a function, $\lambda x.x + 3$ is.
- cannot apply $x + 3$ to an argument: $(x + 3)2$ does not evaluate to 2+3.

Common features of modern types and functions

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

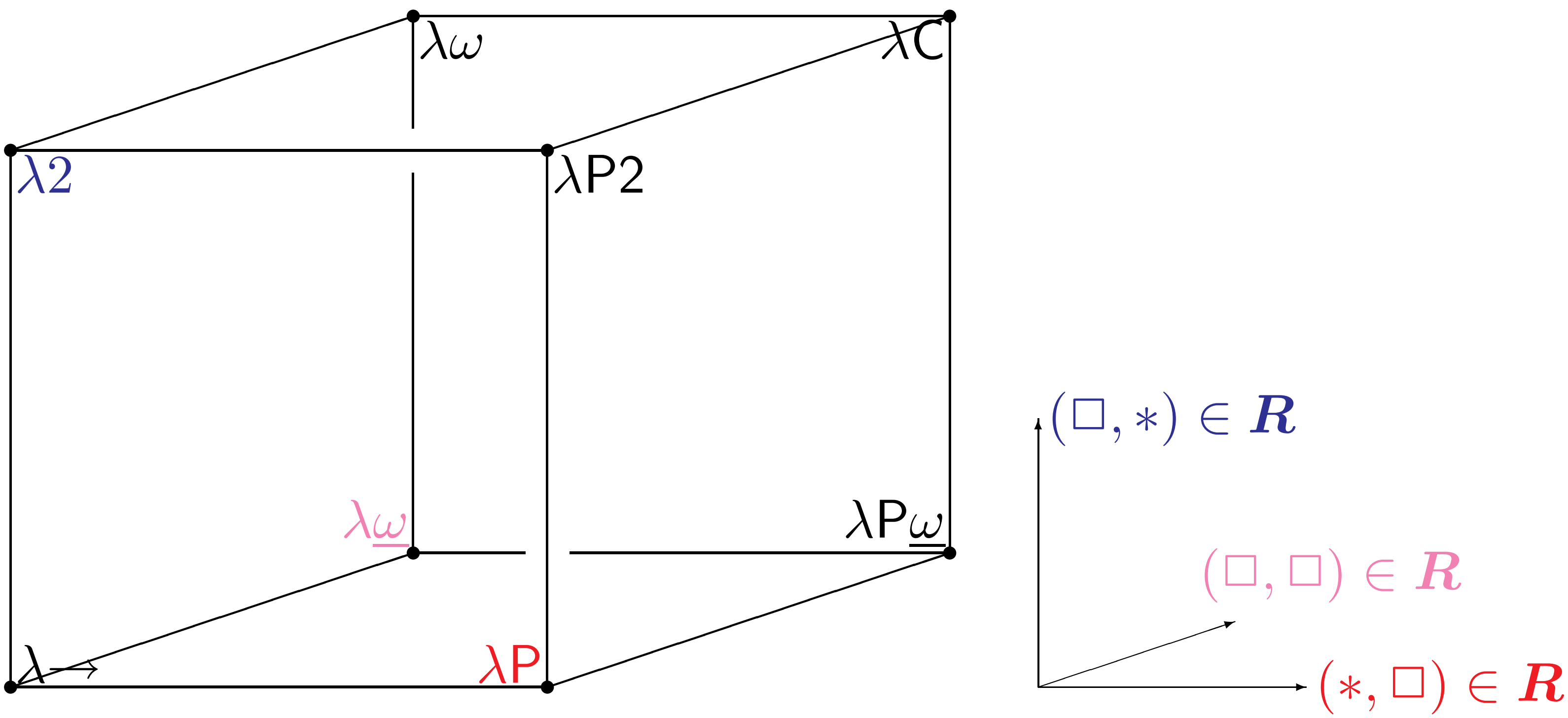
The Barendregt Cube

- Syntax: $A ::= x \mid * \mid \Box \mid AB \mid \lambda x{:}A.B \mid \Pi x{:}A.B$

- Formation rule: $$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$$ if $(s_1, s_2) \in \boldsymbol{R}$

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

The Barendregt Cube



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

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

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

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

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

ML

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

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

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

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

- ML has limited access to the rule $(\Box, *)$ enabling some things from $\lambda 2$ but not all.

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

- We place the type system of ML on our refined Cube (between $\lambda 2$ and $\lambda\underline{\omega}$).

LF

- LF [Harper et al., 1987] is often described as λP of the Barendregt Cube.
- Use of Π-formation rule $(*, \Box)$ is very restricted in the practical use of LF [Geuvers, 1993].
- The only need for a type $\Pi x{:}A.B : \Box$ is when the Propositions-As-Types principle PAT is applied during the construction of the type $\Pi\alpha{:}\mathtt{prop}.*$ of the operator Prf where for a proposition Σ, Prf(Σ) is the type of proofs of Σ.

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

- In LF, this is the only point where the Π-formation rule $(*, \Box)$ is used.
- But, Prf is only used when applied $\Sigma{:}\mathtt{prop}$. We never use Prf on its own.
- This use is in fact based on a parametric constant rather than on Π-formation.
- Hence, the practical use of LF would not be restricted if we present Prf in a parametric form, and use $(*, \Box)$ as a parameter instead of a Π-formation rule.

- We will find a more precise position of LF on the Cube (between $\lambda\!\rightarrow$ and λP).

Parameters: What and Why

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

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

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

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

Automath

- The first tool for mechanical representation and verification of mathematical proofs, AUTOMATH, has a parameter mechanism.

- Mathematical text in AUTOMATH written as a finite list of *lines* of the form:

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

  Here g is a new name, an abbreviation for the expression t of type T and $x_1, \ldots, x_n$ are the parameters of g, with respective types $A_1, \ldots, A_n$.

- Each line introduces a new definition which is inherently parametrised by the variables occurring in the context needed for it.

- Developments of ordinary mathematical theory in AUTOMATH [Benthem Jutting, 1977] revealed that this combined definition and parameter mechanism is vital for keeping proofs manageable and sufficiently readable for humans.

Extending the Cube with parametric constants

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

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

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

- $(s_1, s_2) \in \boldsymbol{P}$ 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 $(*, s_2) \in \boldsymbol{P}$ and $(\square, s_2) \in \boldsymbol{P}$ then combinations of parameters allowed. For example, it is allowed that B_1 has type $*$, whilst B_2 has type $\square$.

The Cube with parametric constants

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

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

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

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

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

Properties of the Refined Cube

- (Correctness of types) If $\Gamma \vdash A : B$ then $(B \equiv \Box$ or $\Gamma \vdash B : S$ for some sort $S)$.
- (Subject Reduction SR) If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$
- (Strong Normalisation) For all $\vdash$-legal terms M, we have $\mathrm{SN}_{\twoheadrightarrow_\beta}(M)$.
- Other properties such as Uniqueness of types and typability of subterms hold.
- $\lambda\mathbf{R}\boldsymbol{P}$ is the system which has Π-formation rules $\boldsymbol{R}$ and parameter rules $\boldsymbol{P}$.
- Let $\lambda\mathbf{R}\boldsymbol{P}$ parametrically conservative (i.e., $(s_1, s_2) \in \boldsymbol{P}$ implies $(s_1, s_2) \in \boldsymbol{R}$).
  - The parameter-free system $\lambda\mathbf{R}$ is at least as powerful as $\lambda\mathbf{R}\boldsymbol{P}$.
  - If $\Gamma \vdash_{\mathbf{R}\boldsymbol{P}} a : A$ then $\{\Gamma\} \vdash_{\boldsymbol{R}} \{a\} : \{A\}$.

Example

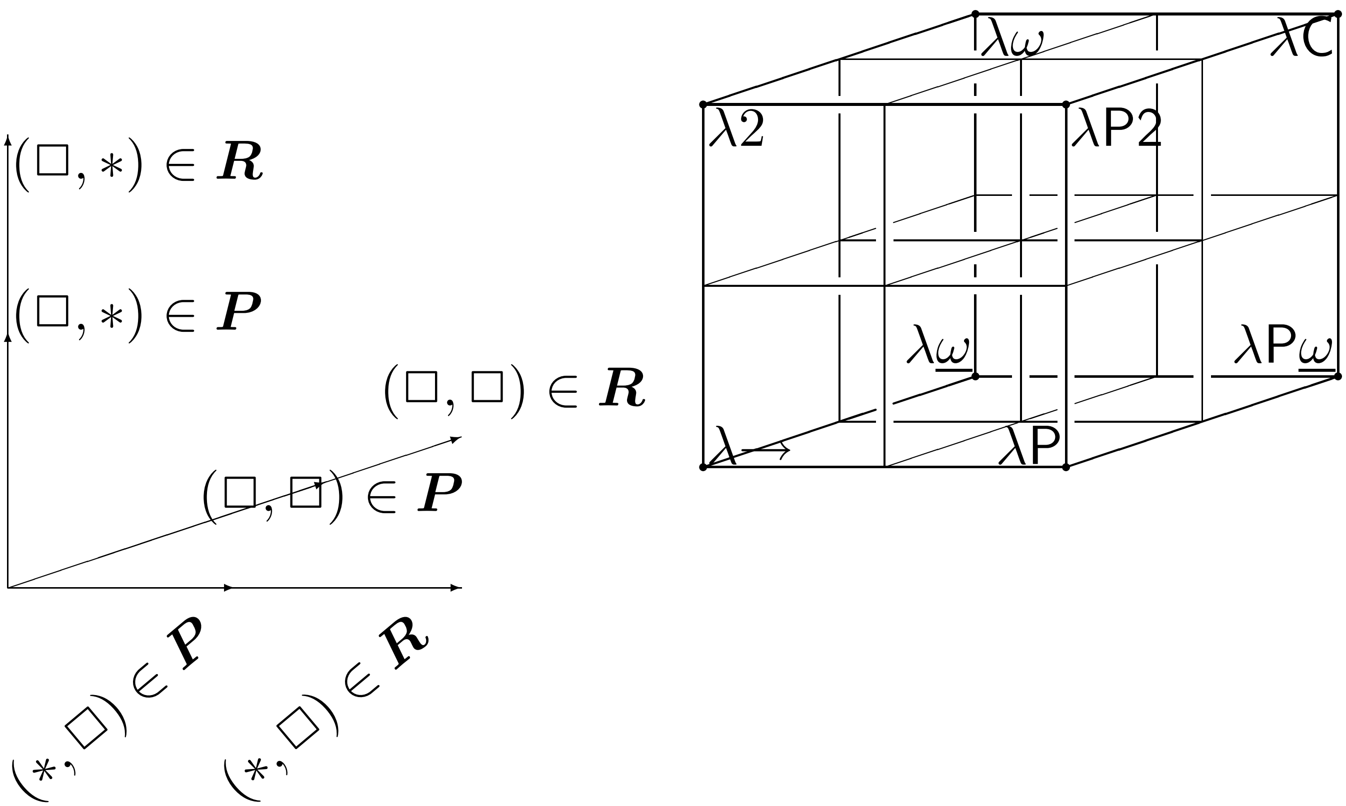
- $\boldsymbol{R} = \{(*, *), (*, \Box)\}$
  $\boldsymbol{P}_1 = \emptyset \qquad \boldsymbol{P}_2 = \{(*, *)\} \qquad \boldsymbol{P}_3 = \{(*, \Box)\} \qquad \boldsymbol{P}_4 = \{(*, *), (*, \Box)\}$
  All $\lambda\boldsymbol{R}\boldsymbol{P}_i$ for $1 \le i \le 4$ with the above specifications are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

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

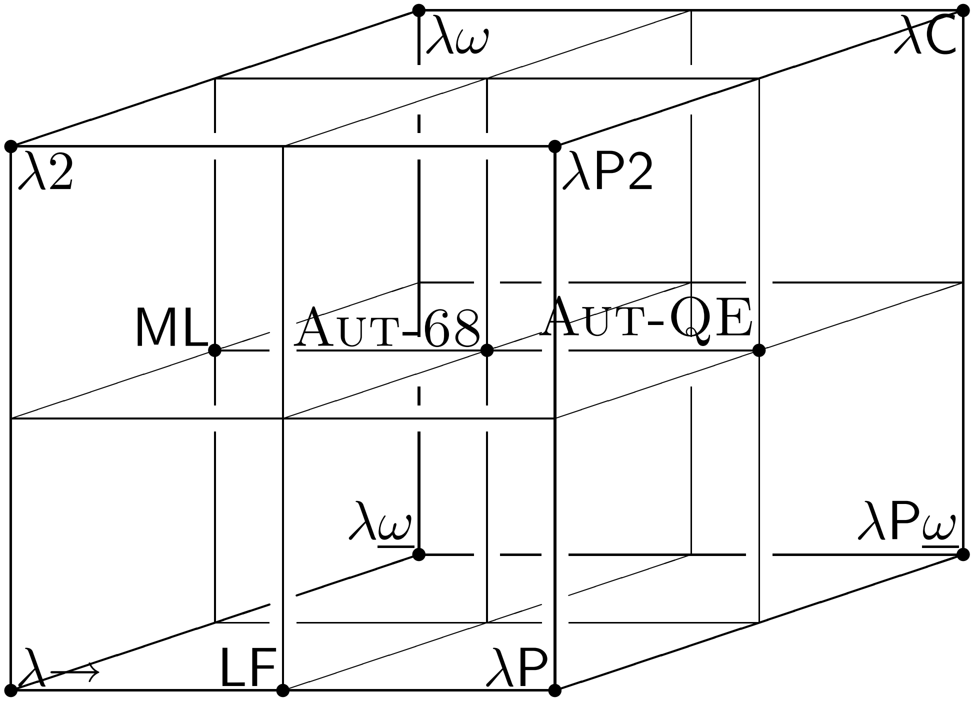
$$\begin{array}{rcl} \alpha & : & *, \\ \mathtt{eq}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha) & : & *, \\ \mathtt{refl}(\mathtt{x}{:}\alpha) & : & \mathtt{eq}(\mathtt{x}, \mathtt{x}), \\ \mathtt{symm}(\mathtt{x}{:}\alpha, \mathtt{y}{:}\alpha, \mathtt{p}{:}\mathtt{eq}(\mathtt{x}, \mathtt{y})) & : & \mathtt{eq}(\mathtt{y},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

  eq not possible in $\lambda\!\to$.

The refined Barendregt Cube



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



Logicians versus mathematicians and induction over numbers

- Logician uses `ind`: `Ind` as proof term for an application of the induction axiom. The type `Ind` can only be described in $\lambda\boldsymbol{R}$ where $\boldsymbol{R} = \{(*,*),(*,\Box),(\Box,*)\}$:

$$\mathtt{Ind} = \Pi p{:}(\mathbb{N}\to *).p0\to(\Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}.pn\to Snm\to pm)\to\Pi n{:}\mathbb{N}.pn \qquad (1)$$

- Mathematician uses `ind` only with $P : \mathbb{N}\to *$, $Q : P0$ and $R : (\Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}.Pn\to Snm\to Pm)$ to form a term $(\mathtt{ind}PQR){:}(\Pi n{:}\mathbb{N}.Pn)$.

- The use of the induction axiom by the mathematician is better described by the parametric scheme (p, q and r are the *parameters* of the scheme):

$$\mathtt{ind}(p{:}\mathbb{N}\to *, q{:}p0, r{:}(\Pi n{:}\mathbb{N}.\Pi m{:}\mathbb{N}.pn\to Snm\to pm)) : \Pi n{:}\mathbb{N}.pn \qquad (2)$$

- The logician's type `Ind` is not needed by the mathematician and the types that occur in 2 can all be constructed in $\lambda\boldsymbol{R}$ with $\boldsymbol{R} = \{(*,*)(*,\Box)\}$.

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).

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

- 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.

Conclusions

- 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.
- 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.

Item Notation/Lambda Calculus à la de Bruijn

- $\mathcal{I}$ translates to item notation:

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

- $(\lambda x.\lambda y.xy)z$ translates to $(z)[x][y](y)x$.
- The *items* are (z), $[x]$, $[y]$ and (y). The last x is the *heart* of the term.
- The *applicator wagon* (z) and *abstractor wagon* $[x]$ occur NEXT to each other.
- The β rule $(\lambda x.A)B \to_\beta A[x := B]$ becomes in item notation:

$$(B)[x]A \to_\beta [x := B]A$$

Redexes in Item Notation

Classical Notation	Item Notation
$(\underline{(\lambda_x.(\lambda_y.\lambda_z.zd)c)b})a$	$(a)\underline{(b)[x]}(c)[y][z](d)z$
$\downarrow_\beta$	$\downarrow_\beta$
$(\underline{(\lambda_y.\lambda_z.zd)c})a$	$(a)\underline{(c)[y]}[z](d)z$
$\downarrow_\beta$	$\downarrow_\beta$
$\underline{(\lambda_z.zd)a}$	$\underline{(a)[z]}(d)z$
$\downarrow_\beta$	$\downarrow_\beta$
ad	$(d)a$

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

Segments, Partners, Bachelors

- 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 $(a)(b)[x](c)[y][z](d)$ is simpler: $\{\{\ \}\{\ \}\}$.

- (a) and $[z]$ are *partners*. (b) and $[x]$ are *partners*. (c) and $[y]$ are *partners*.

- (d) is bachelor.

- A *segment* $\overline{s}$ is *well balanced* when it contains only partnered main items. $(a)(b)[x](c)[y][z]$ is well balanced.

- A segment is bachelor when it contains only bachelor main items.

More on Segments, Partners, and Bachelors

- The *main* items are those at top level.
  In $([y](y)y)[x]x$ the main items are: $([y](y)y)$ and $[x]$.
  $[y]$ and (y) are *not* main items.

- Each main bachelor $[]$ precedes each main bachelor $()$.
  For example, look at: $[u](a)(b)[x](c)[y][z](d)u$.

- Removing all main bachelor items yields a well balanced segment.
  For example from $[u](a)(b)[x](c)[y][z](d)$ we get: $(a)(b)[x](c)[y][z]$.

- Removing all main partnered items yields a bachelor segment $[v_1]\dots[v_n](a_1)\dots(a_m)$.
  For example from $[u](a)(b)[x](c)[y][z](d)$ we get: $[u](d)$.

- If $[v]$ and (b) are partnered in $\overline{s_1}(b)\overline{s_2}[v]\overline{s_3}$, then $\overline{s_2}$ must be well balanced.

Even More on Segments, Partners, and Bachelors

Each non-empty segment $\overline{s}$ has a unique *partitioning* into sub-segments $\overline{s} = \overline{s_0 s_1} \cdots \overline{s_n}$ such that $n \geq 0$,

- $\overline{s_i}$ is not empty for $i \geq 1$,
- $\overline{s_i}$ is well balanced if i is even and is bachelor if i is odd.
- if $\overline{s_i} = [x_1] \cdots [x_m]$ and $\overline{s_j} = (a_1) \cdots (a_p)$ then $\overline{s_i}$ precedes $\overline{s_j}$
- Example: $\overline{s} \equiv [x][y](a)[z][x'](b)(c)(d)[y'][z'](e)$ is partitioned as:
- $\overline{s} \equiv \overbrace{\emptyset}^{\overline{s_0}} \underbrace{[x][y]}_{\overline{s_1}} \overbrace{(a)[z]}^{\overline{s_2}} \underbrace{[x'](b)}_{\overline{s_3}} \overbrace{(c)(d)[y'][z']}^{\overline{s_4}} \underbrace{(e)}_{\overline{s_5}}$

More on Item Notation

- Above discussion and further details of item notation can be found in [Kamareddine and Nederpelt, 1995, 1996].

- Item notation helped greatly in the study of a one-sorted style of explicit substitutions, the λs-style which is related to $\lambda\sigma$, but has certain simplifications [Kamareddine and Ríos, 1995, 1997; Kamareddine and Ríos, 2000].

- For explicit substitution in item notation see [Kamareddine and Nederpelt, 1993]

Canonical Forms

- Nice canonical forms look like:

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

- classical:

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

- For example, a canonical form of:

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

is

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

Some Helpful Rules for reaching canonical forms

Name	In Classical Notation	In Item Notation
(θ)	$((\lambda_x.N)P)Q$ $\downarrow$ $(\lambda_x.NQ)P$	$(Q)(P)[x]N$ $\downarrow$ $(P)[x](Q)N$
(γ)	$(\lambda_x.\lambda_y.N)P$ $\downarrow$ $\lambda_y.(\lambda_x.N)P$	$(P)[x][y]N$ $\downarrow$ $[y](P)[x]N$
(γ_C)	$((\lambda_x.\lambda_y.N)P)Q$ $\downarrow$ $(\lambda_y.(\lambda_x.N)P)Q$	$(Q)(P)[x][y]N$ $\downarrow$ $(Q)[y](P)[x]N$
(g)	$((\lambda_x.\lambda_y.N)P)Q$ $\downarrow$ $(\lambda_x.N[y:=Q])P$	$(Q)(P)[x][y]N$ $\downarrow$ $(P)[x][y:=Q]N$

A Few Uses of Generalised Reduction and Term Reshuffling

- Regnier [1992] uses term reshuffling and generalized reduction in analyzing perpetual reduction strategies.
- Term reshuffling is used in [Kfoury et al., 1994; Kfoury and Wells, 1994] in analyzing typability problems.
- [Nederpelt, 1973; de Groote, 1993; Kfoury and Wells, 1995] use generalised reduction and/or term reshuffling in relating SN to WN.
- [Ariola et al., 1995] uses a form of term-reshuffling in obtaining a calculus that corresponds to lazy functional evaluation.
- [Kamareddine and Nederpelt, 1995; Kamareddine et al., 1999, 1998; Bloo et al., 1996] shows that they could reduce space/time needs.
- [Kamareddine, 2000] shows various strong properties of generalised reduction.

Obtaining Canonical Forms

θ-nf:		$()[\,]$-pairs mixed with bach. $[\,]$s $(A_1)[x][y][z](A_2)[p]\cdots$	bach. $()$s $(B_1)(B_2)\cdots$	end var x
γ-nf:	bach. $[\,]$s $[x_1][x_2]\cdots$	$()[\,]$-pairs mixed with bach. $()$s $(B_1)(A_1)[x](B_2)\cdots$		end var x
θ-γ-nf:	bach. $[\,]$s $[x_1][x_2]\cdots$	$()[\,]$-pairs $(A_1)[y_1](A_2)[y_2]\ldots(A_m)[y_m]$	bach. $()$s $(B_1)(B_2)\ldots$	end var x
γ-θ-nf:	bach. $[\,]$s $[x_1][x_2]\cdots$	$()[\,]$-pairs $(A_1)[y_1](A_2)[y_2]\ldots(A_m)[y_m]$	bach. $()$s $(B_1)(B_2)\ldots$	end var x

Example

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

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

Classes of terms modulo reductional behaviour

- $\rightarrow_\theta$ and $\rightarrow_\gamma$ are SN and CR. Hence θ-nf and γ-nf are unique.

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

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

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

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

- When $B \in [A]$, we write that $B \approx_{\mathtt{equi}} A$.

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

- Define $\mathrm{CCF}(A)$ as $\{A' \text{ in canonical form} \mid A' =_p \theta(\gamma(A))\}$.

Reduction based on classes

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

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

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

- If $A \leadsto_\beta B$ then forall $A' \approx_{\texttt{equi}} A$, forall $B' \approx_{\texttt{equi}} B$, we have $A' \leadsto_\beta B'$.

Properties of reduction modulo classes

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

Using Item Notation in Type Systems

- Now, all items are written inside $()$ instead of using $()$ and $[]$.
- $(\lambda_x.x)y$ is written as: $(y\delta)(\lambda_x)x$ instead of $(y)[x]x$.
- $\Pi_{z:*}.(\lambda_{x:z}.x)y$ is written as: $(*\Pi_z)(y\delta)(z\lambda_x)x$.

The Barendregt Cube in item notation and class reduction

- The formulation is the same except that terms are written in item notation:
- $\mathcal{T} = * \mid \Box \mid V \mid (\mathcal{T}\delta)\mathcal{T} \mid (\mathcal{T}\lambda_V)\mathcal{T} \mid (\mathcal{T}\Pi_V)\mathcal{T}$.
- The typing rules don't change although we do class reduction $\leadsto_\beta$ instead of normal β-reduction $\to_\beta$.
- The typing rules don't change because $=_\beta$ is the same as $\approx_\beta$.

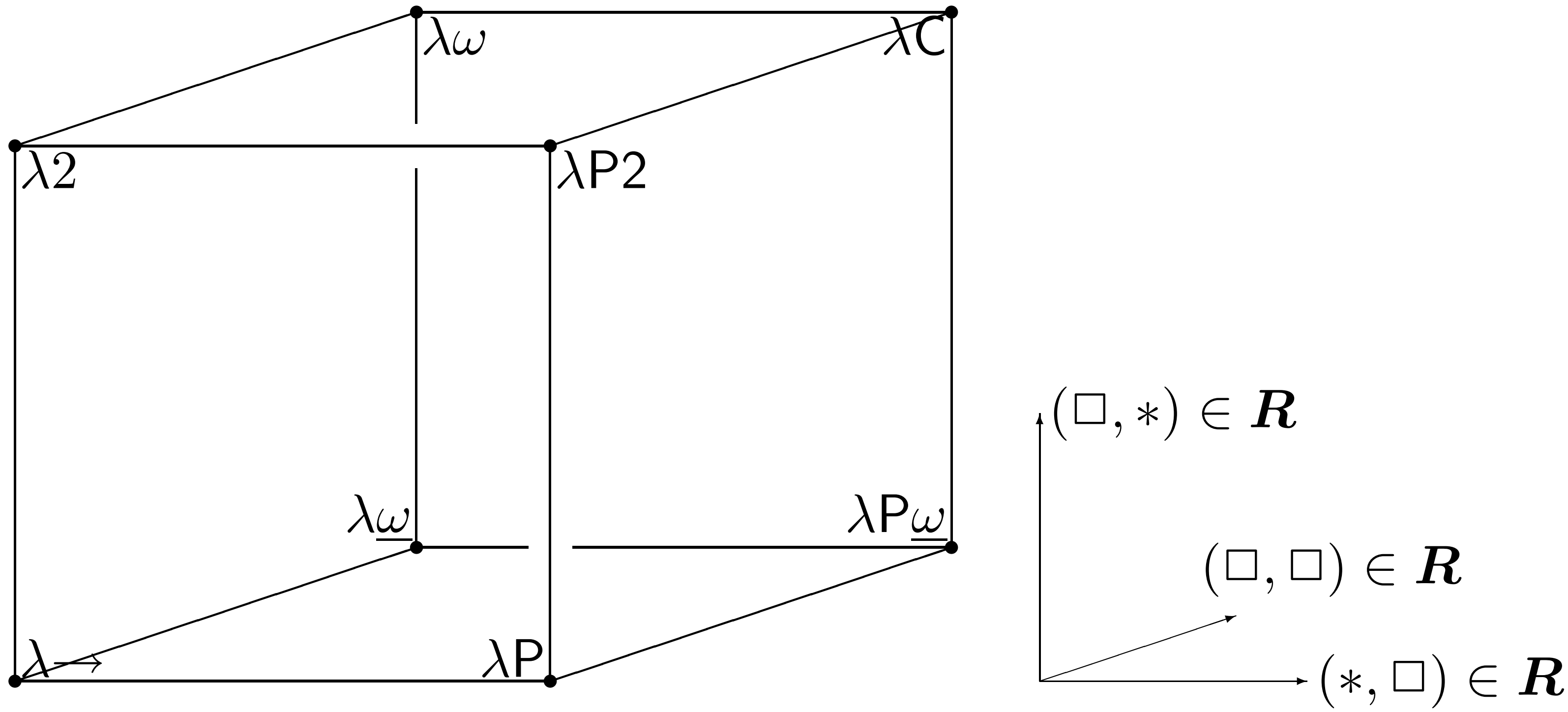


Figure 1: The Barendregt Cube

Subject Reduction fails

- Most properties including SN hold for all systems of the cube extended with class reduction. However, SR only holds in $\lambda_{\rightarrow}$ $(*, *)$ and $\lambda\underline{\omega}$ $(\Box, \Box)$.
- SR fails in λP $(*, \Box)$ (and hence in $\lambda P2, \lambda P\underline{\omega}$ and λC). Example in paper.
- SR also fails in $\lambda 2$ $(\Box, *)$ (and hence in $\lambda P2, \lambda\omega$ and λC):

Why does Subject Reduction fails

- $(y'\delta)(\beta\delta)(*\lambda_\alpha)(\alpha\lambda_y)(y\delta)(\alpha\lambda_x)x \rightsquigarrow_\beta (\beta\delta)(*\lambda_\alpha)(y'\delta)(\alpha\lambda_x)x$.
- $(\lambda_{\alpha:*}.\lambda_{y:\alpha}.(\lambda_{x:\alpha}.x)y)\beta y' \rightsquigarrow_\beta (\lambda_{\alpha:*}.(\lambda_{x:\alpha}.x)y')\beta$
- $\beta : *, y' : \beta \vdash_{\lambda 2} (\lambda_{\alpha:*}.\lambda_{y:\alpha}.(\lambda_{x:\alpha}.x)y)\beta y' : \beta$
- Yet, $\beta : *, y' : \beta \not\vdash_{\lambda 2} (\lambda_{\alpha:*}.(\lambda_{x:\alpha}.x)y')\beta : \tau$ for any τ.
- the information that $y' : \beta$ has replaced $y : \alpha$ is lost in $(\lambda_{\alpha:*}.(\lambda_{x:\alpha}.x)y')\beta$.
- But we need $y' : \alpha$ to be able to type the subterm $(\lambda_{x:\alpha}.x)y'$ of $(\lambda_{\alpha:*}.(\lambda_{x:\alpha}.x)y')\beta$ and hence to type $\beta : *, y' : \beta \vdash (\lambda_{\alpha:*}.(\lambda_{x:\alpha}.x)y')\beta : \beta$.

Solution to Subject Reduction: Use "let expressions/definitions"

- Definitions/let expressions are of the form: let $x : A \ = B$ and are added to contexts exactly like the declarations $y : C$.

- (def rule) $$\frac{\Gamma, \mathsf{let}\ x : A \ = B \vdash^{\mathsf{c}} C : D}{\Gamma \vdash^{\mathsf{c}} (\lambda_{x:A}.C)B : D[x := A]}$$

- we define $\Gamma \vdash^{\mathsf{c}} \cdot =_{\mathtt{def}} \cdot$ to be the equivalence relation generated by:
  - if $A =_\beta B$ then $\Gamma \vdash^{\mathsf{c}} A =_{\mathtt{def}} B$
  - if let $x : M \ = N$ is in Γ and if B arises from A by substituting one particular occurrence of x in A by N, then $\Gamma \vdash^{\mathsf{c}} A =_{\mathtt{def}} B$.

The (simplified) Cube with definitions and class reduction

(axiom) (app) (abs) *and* (form) are unchanged.

(start) $$\frac{\Gamma \vdash^{c} A : s}{\Gamma, x{:}A \vdash^{c} x : A} \qquad \frac{\Gamma \vdash^{c} A : s \qquad \Gamma \vdash^{c} B : A}{\Gamma,\ \mathsf{let}\ x : A\ = B \vdash^{c} x : A}$$ x fresh

(weak) $$\frac{\Gamma \vdash^{c} D : E \quad \Gamma \vdash^{c} A : s}{\Gamma, x{:}A \vdash^{c} D : E} \qquad \frac{\Gamma \vdash^{c} A : s \quad \Gamma \vdash^{c} B : A \quad \Gamma \vdash^{c} D : E}{\Gamma,\ \mathsf{let}\ x : A\ = B \vdash^{c} D : E}$$ x fresh

(conv) $$\frac{\Gamma \vdash^{c} A : B \qquad \Gamma \vdash^{c} B' : S \qquad \Gamma \vdash^{c} B =_{\mathtt{def}} B'}{\Gamma \vdash^{c} A : B'}$$

(def) $$\frac{\Gamma, \mathsf{let}\ x : A\ = B \vdash^{c} C : D}{\Gamma \vdash^{c} (\lambda_{x:A}.C)B : D[x := A]}$$

Definitions solve subject reduction

1. $\beta : *, y' : \beta, \text{ let } \alpha : * = \beta \quad \vdash^{c} y' : \beta$

2. $\beta : *, y' : \beta, \text{ let } \alpha : * = \beta \quad \vdash^{c} \alpha =_{\texttt{def}} \beta$

3. $\beta : *, y' : \beta, \text{ let } \alpha : * = \beta \quad \vdash^{c} y' : \alpha$ (from 1 and 2)

4. $\beta : *, y' : \beta, \text{ let } \alpha : * = \beta, \text{ let } x : \alpha = y' \quad \vdash^{c} x : \alpha$

5. $\beta : *, y' : \beta, \text{ let } \alpha : * = \beta \quad \vdash^{c} (\lambda_{x:\alpha}.x)y' : \alpha[x := y'] = \alpha$

$$\beta : *, y' : \beta \quad \vdash^{c} \quad (\lambda_{\alpha:*}.(\lambda_{x:\alpha}.x)y')\beta : \alpha[\alpha := \beta] = \beta$$

Properties of the Cube with definitions and class Reduction

- $\vdash^c$ is a generalisation of $\vdash$: If $\Gamma \vdash A : B$ then $\Gamma \vdash^c A : B$.

- Equivalent terms have same types:
  If $\Gamma \vdash^c A : B$ and $A' \in [A]$, $B' \in [B]$ then $\Gamma \vdash^c A' : B'$.

- Subject Reduction for $\vdash^c$ and $\rightsquigarrow\!\!\!\twoheadrightarrow_\beta$:
  If $\Gamma \vdash^c A : B$ and $A \rightsquigarrow\!\!\!\twoheadrightarrow_\beta A'$ then $\Gamma \vdash^c A' : B$.

- Unicity of Types for $\vdash^c$:
  - If $\Gamma \vdash^c A : B$ and $\Gamma \vdash^c A : B'$ then $\Gamma \vdash^c B =_{\mathtt{def}} B'$
  - If $\Gamma \vdash^c A : B$ and $\Gamma \vdash^c A' : B'$ and $\Gamma \vdash^c A =_\beta A'$ then $\Gamma \vdash^c B =_{\mathtt{def}} B'$.

- Strong Normalisation of $\rightsquigarrow\!\!\!\twoheadrightarrow_\beta$:
  In the Cube, every legal term is strongly normalising with respect to $\rightsquigarrow\!\!\!\twoheadrightarrow_\beta$.

typed λ-calculus with single binder

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

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

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

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

Polymorphism: the typed λ-calculus after Church

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

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

  We give F the type:

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

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

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

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

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

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

  to a rule which resembles Π:

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

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

- In the literature there are many studies (like de Bruijn's Automath) which shows that the rule Π is useful.

- It seems that the development of type theory is taking us more and more towards adopting a similar roles for the binders λ et Π. Did we really need to separate them? I believe that the separation is artificial.

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

- Is the passage from a type system with two binders to a type system with one single binder as necessary as the passage from simple to polymorphic types?

- I think that from the start, we should not have been limited to simple types and we should not have distinguished terms and types.

Notation of Modern type systems

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

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

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

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

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

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

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

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

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

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

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

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

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

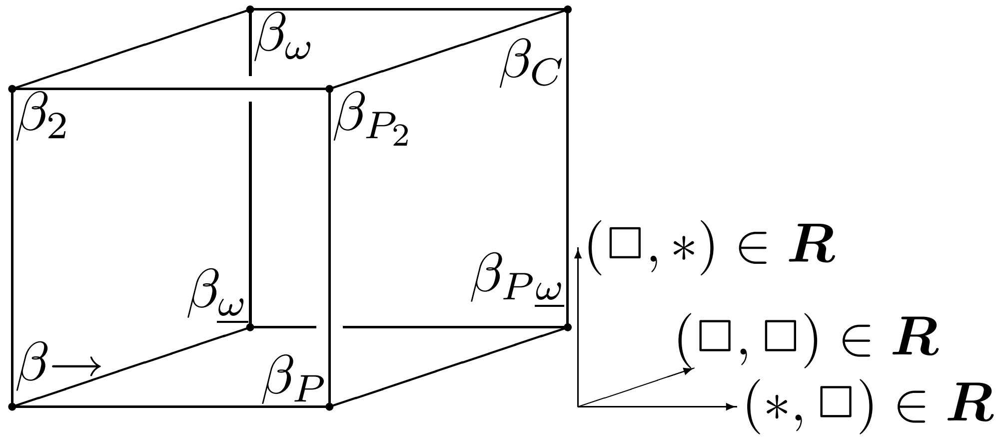


Figure 2: The β-cube of Barendregt

Our example in the system F of Girard

- If $x \not\in FV(B)$ we write $A \to B$ instead of $\Pi_{x:A}.B$.

- $\alpha : *, f : \alpha \to \alpha \vdash \lambda_{x:\alpha}.f(f(x)) : \alpha \to \alpha : *$
(we need the rule $(*, *)$).

- $\alpha : * \vdash \lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f(x)) : (\alpha \to \alpha) \to (\alpha \to \alpha) : *$
(we need the rule $(*, *)$).

- $\vdash \lambda_{\alpha:*}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f(x)) : \Pi_{\alpha:*}.(\alpha \to \alpha) \to (\alpha \to \alpha) : *$
(we need the rule $(\Box, *)$).

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

(axiom) (start) (weak) (Π) (λ) (appl)

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

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

(axiom)	(start) (weak) (Π) (λ)
(conv$_{\beta\Pi}$)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{\beta\Pi} B'}{\Gamma \vdash A : B'}$
(newappl)	$\dfrac{\Gamma \vdash F : \Pi_{x:A}.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : (\Pi_{x:A}.B)a}$

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

(axiom) (start) (weak)

(♭₁) $$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2 \quad (s_1, s_2) \in \boldsymbol{R}}{\Gamma \vdash \flat_{x:A}.B : s_2}$$

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

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

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

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

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

6 desirable properties of a type system with reduction r

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

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

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

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

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

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

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

- Lemma:
  - $\Gamma \vdash_\beta A : B$ iff $\Gamma \vdash_\pi A : B$
  - If $\Gamma \vdash_\beta A : B$ then $\Gamma \vdash_{\pi_i} A : B$.
  - If $\Gamma \vdash_{\pi_i} A : B$ then $\Gamma \vdash_\beta A : [B]_\Pi$
    where $[B]_\Pi$ is the Π-normal form of B.

- Lemma: The β-cube and the π-cube satisfy the six properties that are desirable for type systems.

The π_i-cube

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

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

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

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

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

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

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

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

The isomorphism

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

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

The $\flat$-cube

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

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

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

- Let $\mathsf{SN}_{\to_\flat}(B_1)$ and $\mathsf{SN}_{\to_\flat}(B_2)$. We say that $B_1 \stackrel{\diamond}{=}_\flat B_2$ iff $\mathrm{nf}_\flat(B_1) \equiv \flat^{i:1..n_1}_{x_i:F_i}.B$ and $\mathrm{nf}_\flat(B_2) \equiv \flat^{i:1..n_2}_{x_i:F_i}.B$, where $n_1, n_2 \geq 0$.

- *Lemma (MOT)* If $\Gamma \vdash_\flat A_1 : B_1$ and $\Gamma \vdash_\flat A_2 : B_2$ and $A_1 =_\flat A_2$, then $B_1 \stackrel{\diamond}{=}_\flat B_2$.

- Hence, the $\flat$-cube is elegant, and has all the desirable properties (we do not want UT1, MOT is enough).

Example

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

- In the β-cube we have:

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

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

Conclusion

- If the type system can do all that we want (as we have seen in the $\flat$-cube), why separate the levels of types and terms?

- Terms and types are not different. The only difference is the relation that exists between them. In $A : B$, the A on the left represents a term, the B on the right represents a type.

- Even if you do not want to use this system, its existence is already of significance.

- Its presence shows that the division between terms and types is artificial. There was never a need for that division.

- We must also explore the idea that proofs and propositions are also the same thing and it is worth it to study the implications on logic and programming in general.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

Kamareddine, Laan, and Nederpelt. *The A Modern Perspective on Type Theory From its Origins Until Today*, volume 29 of *Applied Logic Series*. Kluwer Academic Publishers.

F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-styles of explicit substitutions. *J. Logic Comput.*, 10(3): 399–431, 2000.

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

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

Fairouz Kamareddine and Rob Nederpelt. On stepwise explicit substitution. *Int'l J. Foundations Comput. Sci.*, 4(3): 197–240, 1993.

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

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

Fairouz Kamareddine and Alejandro Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *J. Funct. Programming*, 7(4):395–420, 1997.

Fairouz Kamareddine and Alejandro Ríos. A λ-calculus à la de Bruijn with explicit substitution. In *7th Int'l Symp. Prog. Lang.: Implem., Logics & Programs, PLILP '95*, volume 982 of *Lecture Notes in Computer Science*, pages 45–62. Springer, 1995.

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

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

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

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

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

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

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

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

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

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

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

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.