Linguas y Modeles por el Formalisatione y el Automation del Matemáticas y el Informatica

Fairouz Kamareddine (Universidad de Heriot-Watt, Edimbourgo, RU)

Noviembre 2009

The computer revolution

- In less than a half a century, computers have revolutionised the way we all live.
- Google, Wikipedia, and other information and search engines have changed the way we store and exchange information.
- Computerisation also enables excellent collaborations between different disciplines (think of Bio-Informatics) and enables new discoveries in different disciplines.
- This computerisation of information is only at its beginning. We need a lot of investments in research methods that enable faster, correct, and efficient information storage and retrieval.
- Information here means every aspect of information (mathematical, medical, social, educational, law, etc).
- Calculators process numbers, computers process information.

The languages of Mathematics

Usually, mathematicians ignore formal logic and write mathematics using a certain language style which we call CML. Advantages of CML:

- *Expressivity:* We can express all sorts of notions.
- *Acceptability:* CML is accepted by most mathematicians.
- *Traditionality:* CML exists since very long and has been refined with the time.
- *Universality:* CML is used everywhere.
- *Flexibility:* With CML we can descibe several branches of mathematics (including new ones).

The Disadvantage of CML

- *Informal and ambiguous:* CML is based on natural language.
- *Incomplete:* The author counts on the intuition of the reader.
- *Not easy to automate*
- Initially, people were worried about ambiguity and incompleteness. Automation was the outcome of precision.

Precision and Automation

- The needs to become more precise were strongly felt in the 19^{th} century (e.g., think of the problems in Analysis).

- Many of these problems were solved by the work of Cauchy (e.g., his precise definition of convergence in his Cours d'Analyse).

- Also number systems became more precise with the definition of real numbers of Dedekind.

- Cantor started the formalisation of set theory and contributed to number theory.

Logic, functions, λ-calculus and type theory

- Frege was frustrated by the informalities of $\textsc{Cml}$.
- *The general definition of function* was key to his *formalisation of logic* (1879).
- *The application of a function to itself* $f(x) = \neg x(x)$ was key to *Russell's paradox* (1902). See [Kamareddine et al., 2002].
- To eliminate the paradox, Russell controlled the application of a function to an argument by his *theory of types*.
- Russell (1908) gave the first theory of types $\textsc{rtt}$. Russell and Whitehead used $\textsc{rtt}$ in *Principia Mathematica* (1910–1912).
- *simple theory of types* ($\textsc{stt}$): Ramsey (1926), Hilbert and Ackermann (1928).
- In 1928, Church wanted to write a language of *functions and logic*:
  $\Lambda ::= \mathcal{V} | (\Lambda\Lambda) | \lambda\mathcal{V}.\Lambda | \neg\Lambda | \forall\mathcal{V}.\Lambda(\mathcal{V})$

- The combination of functions and logic was paradoxical.
  The problem was not with $\neg$, but $\forall$.

- In 1932, he eliminated logic and his λ-calcul became a calculus of functions.

- In 1940, he added logic and used types to eliminate the paradoxes. *Simply typed λ-calculus* $\lambda\!\to\, = \lambda$-calculus $+$ STT (1940).

- The need for precision in the 19th and 20th century led to the development of computation theory (the invention of the Turing *machine*, the invention of the *language* of computability and the *logic* of computability and decidability).

- My interests are in the machine, language and logic of computation and the computerisation of information.

I started from the λ-calculus (the language) and its model

- It is known that $|A| < |A \mapsto A|$ and even $|A| < |A \mapsto \{0,1\}|$ (Cantor).

- In 1969, Dana Scott wanted to show the non-existence of models of the λ-calculus.

- Contrary to what he wanted, he constructed a model of the λ-calculus where $D \sim (D \longrightarrow D)$, where $(D \longrightarrow D)$ is the set of continuous functions from D to D and D is the fixed point of a continuous construction.

- Since Scott domains do not permit logic (they are models of the function system, the λ-calculus), Peter Aczel intoduced in 1980 "Frege structures" which are models of both functions and logic.

- My PhD thesis introduced λ-calcluli based on Frege structure, established soundness and completeness and used these calculi to study pardoxical and self-referential sentences and to computerise recursion and fixed point operators.

- This is important for programming. You do want to be as expressive as possible while having a full control and knowledge of when programs terminate/not.
- LISP is very expressive but you cannot guarantee termination of your programs.
- A number of attempts have been made at finding new programming languages that have the nice properties (of controling termination, efficiency of time and space, confluence, etc).
- Restricting the expressiveness of the programming language is not good. Our programs must be able to express as much as possible.
- My work guarantees that we can have as much expressivity as possible while being able to handle non terminating programs.

The next step

- Then, I realised that even though the λ-calculus represents exactly the computable functions, it is not a sufficient framework to represent computerisation and to answer efficieny and space issues or to model programming languages.

- I was also interested in the formalisation and automatisation of mathematics in particular and information in general.

- Let us repeat the essence of the λ-calculus:
  Syntax: $\Lambda ::= \mathcal{V}|(\Lambda\Lambda)|\lambda\mathcal{V}.\Lambda$.
  Computation rule: $(\lambda x.A)B \to_\beta A[x := B]$

The lambda Calculus à la de Bruijn (item notation) [Kamareddine and Nederpelt, 1995, 1996]

$'$: Classical Notation	$\mapsto$	Notation of de Bruijn
x	$\mapsto$	x
$\lambda x.B$	$\mapsto$	$[x]B'$
AB	$\mapsto$	$(B')A'$

Example: $(\lambda x.\lambda y.xy)z \quad \mapsto \quad (z)[x][y](y)x$

- In the *train* $(z)[x][y](y)$, the *wagons* are (z), $[x]$, $[y]$ and (y).
- The last x in $(z)[x][y](y)x$ is the *heart* of the term.
- The *application wagon* (z) and the *wagon of abstraction* $[x]$ are *next to each other*.
- The $\beta rule$ $\quad (\lambda x.A)B \to_\beta A[x := B]$ becomes: $(B)[x]A \to_\beta [x := B]A$

Reduction 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	$(d)a$

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

Each wagon has a *partener* except (d) which is *bachelor*.
The parenthesis $((\lambda_x.(\lambda_y.\lambda_z.--)c)b)a)$, are '$[_1\ [_2\ [_3\ ]_2\ ]_1\ ]_3$', where '$[_i$' and '$]_i$' go together.
The parenthesis of $(a)(b)[x](c)[y][z]$ are much simpler: $[[\,][\,]]$.

Generalised Reductions

- $((\lambda_x.(\underline{\lambda_y.\lambda_z.zd)c})b)a \to_\beta ((\lambda_x.\{\lambda_z.zd\}[y := c])b)a$

  $(a)(b)[x](c)[y][z](d)z \to_\beta (a)(b)[x][y := c][z](d)z$

- $(\underline{(\lambda_x.(\lambda_y.\lambda_z.zd)c)b})a \to_\beta \{(\lambda_y.\lambda_z.zd)c\}[x := b]a$

  $(a)(b)[x](c)[y][z](d)z \to_\beta$ (a)[x:=b](c)[y][z](d)z

- $(a)(b)[x](c)[y][z](d)z \hookrightarrow_\beta (b)[x](c)[y][z := a](d)z$

Some notions of reduction in the literature

Name	In Classical Notation	In de Bruijn's 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 these reductions/term reshuffling

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

Especially

- Reducing SN to WN: If we want to know that a program always terminate, it is enough to show that it terminates once only. (Excellent simplification).
- We are now able to analyse the efficieny of time and space of our programs. Huge investments are made by companies to create faster, shorter and more efficient programs.
- Controling the reduction strategy so that we do as little work as possible is also important to save resources. Lazy evaluation of programs is a good step here.
- Other reduction strategies are important. In programming, there is a tradeoff between termination and efficiency.
- With our generalised strategies (the most generalised that exist so far), we are able to choose the right strategy for the right task.

Canonical Forms [Kamareddine et al., 2001]

- Different programs may lead to the same value and do the same thing. Can we find the representative program of such a collection?

- YES!

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

- In [Regnier, 1994] and [Kfoury and Wells, 1995]

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

- For example, the canonical form of:

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

  is

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

How to obtain canonical forms

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

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

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

$\theta(\gamma(A))$ and $\gamma(\theta(A))$ are both in *canonical form*

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

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

Reduction based on the classes of canonical forms [Kamareddine and Bloo, 2002]

- Given a program, can we find all the programs that are equivalent to it in terms of termination, efficiency, value, etc?

- YES!

- Let us define $[A] = \{B \mid \theta(\gamma(A)) =_p \theta(\gamma(B))\}$.

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

- We can now even defrine the most ever generalised notion of reduction.
  $A \leadsto_\beta B \quad$ iff $\quad \exists A' \in [A].\exists B' \in [B].\ A' \rightarrow_\beta B'$ (with compatibility)

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

- $\rightarrow_\beta \subset \rightarrow_g \subset \leadsto_\beta \subset =_\beta = \approx_\beta$.

The most generalised form of reduction has the important properties

- $\rightsquigarrow\!\!\!\twoheadrightarrow_\beta$ *is CR:* If $A \rightsquigarrow\!\!\!\twoheadrightarrow_\beta B$ and $A \rightsquigarrow\!\!\!\twoheadrightarrow_\beta C$, then $\exists D$: $B \rightsquigarrow\!\!\!\twoheadrightarrow_\beta D$ and $C \rightsquigarrow\!\!\!\twoheadrightarrow_\beta D$.
- Let $r \in \{\rightarrow_\beta, \rightsquigarrow_\beta\}$. If $A \in SN_r$ and $A' \in [A]$ then $A' \in SN_r$.
- $A \in SN_{\rightsquigarrow_\beta}$ iff $A \in SN_{\rightarrow_\beta}$.
- We have now a general and powerful way to classify programs according to their evaluation behaviour and termination.
- Numerous research on guaranteeing separate properties of programs has been now generalised and unified into one framework: that of generalised reduction modulo classes.

So far, we extended the λ-calculus without logic

- We need to add logic to have programs that can describe and do what we want.
- Can we add logic to our new λ-calculus with generalised reduction modulo classes?
- Would we still be able to control termination and efficiency?
- Would our new programs be safe and correct?
- Would the final value of the program still be independent of the evaluation path of the program?
- Yes, we can do all this. Recall that to add logic we need types (to avoid the paradoxes).

Simple types: Pascal

- Let $f : \mathbb{N} \to \mathbb{N}$ and $g_f : \mathbb{N} \to \mathbb{N}$ 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	$(\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B})$

- *The problems* in RTT and STT led to the creation of *different type systems*, each with its own *functional abstraction power*.
- *8 important λ-calculi* 1940–1988 were unified in the *cube of Barendregt*.

Polymorphism: the typed λ-calculus after Church: ML, Java

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

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

  We give F the type:

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

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

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

The cube of Barendregt

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

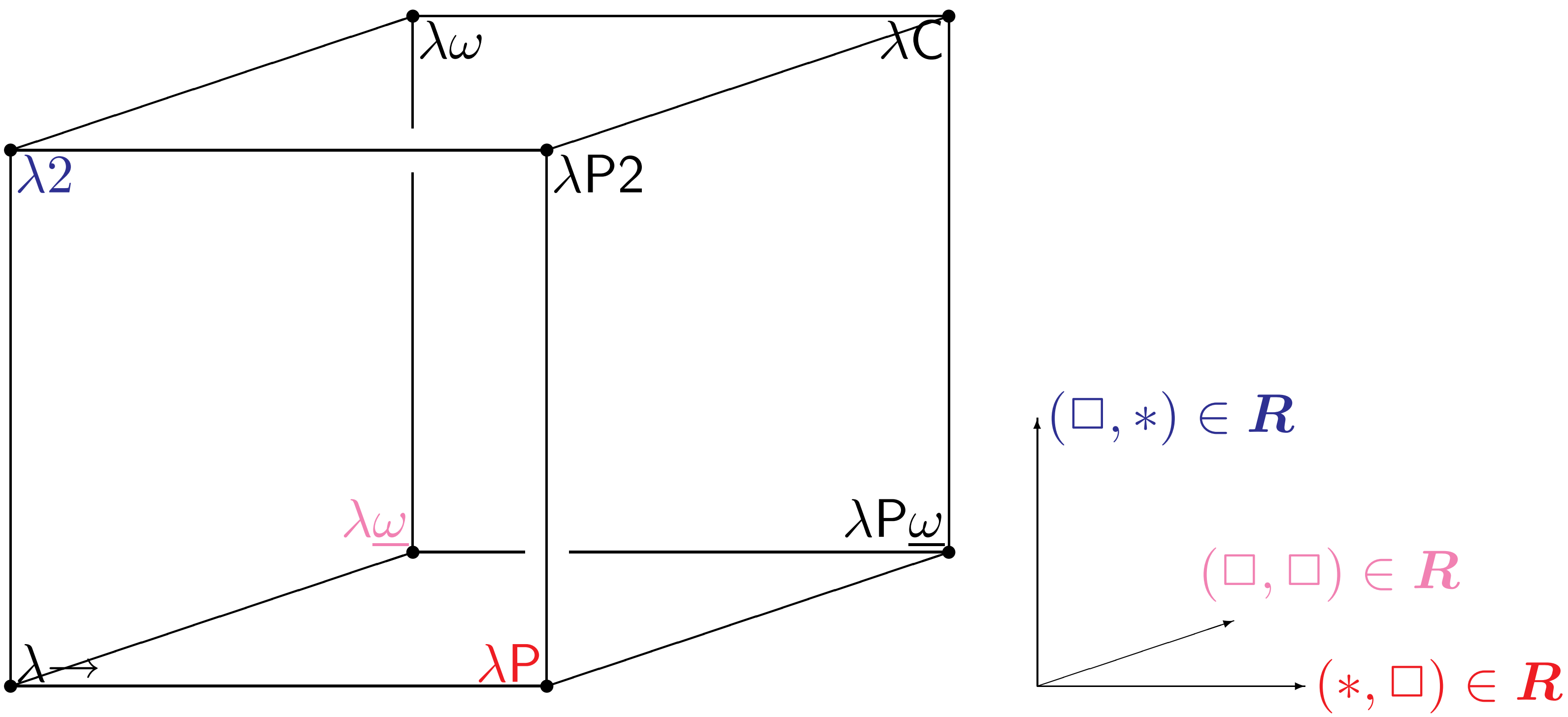
- Formation rule: $\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in \boldsymbol{R}$

	Simple	Poly-morphic	Depend-ent	Constr-uctors	Related system	Refs.
$\lambda{\to}$	$(*, *)$				λ^τ	[Church, 1940; Barendregt, 1984]
$\lambda 2$	$(*, *)$	$(\Box, *)$			F	[Girard, 1972; Reynolds, 1974]
λP	$(*, *)$		$(*, \Box)$		AUT-QE, LF	[Bruijn, 1968; Harper et al., 1987]
$\lambda\underline{\omega}$	$(*, *)$			$(\Box, \Box)$	POLYREC	[Renardel de Lavalette, 1991]
λP2	$(*, *)$	$(\Box, *)$	$(*, \Box)$			[Longo and Moggi, 1988]
$\lambda\omega$	$(*, *)$	$(\Box, *)$		$(\Box, \Box)$	Fω	[Girard, 1972]
λP$\underline{\omega}$	$(*, *)$		$(*, \Box)$	$(\Box, \Box)$		
λC	$(*, *)$	$(\Box, *)$	$(*, \Box)$	$(\Box, \Box)$	CC	[Coquand and Huet, 1988]

The typing rules

(axiom)	$\langle\rangle \vdash * : \Box$
(start)	$\dfrac{\Gamma \vdash A : s \quad x \notin \mathrm{DOM}\,(\Gamma)}{\Gamma, x{:}A \vdash x : A}$
(weak)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s \quad x \notin \mathrm{DOM}\,(\Gamma)}{\Gamma, x{:}C \vdash A : B}$
(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)	$\dfrac{\Gamma \vdash F : \Pi_{x:A}.B \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x{:=}a]}$

The Barendregt cube



Our example in the system F of Girard

- If $x \notin FV(B)$ we write $A \to B$ instead of $\Pi_{x:A}.B$.

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

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

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

The programming language ML

- The passage from simple types ($\lambda_{\rightarrow}$) to polymorphic types ($\lambda 2$) was dictated by many needs including programming ones.

- $\lambda 2$ was invented by John Reynolds at Carnegie Mellon (and independently by Jean-Yves Girard at Paris).

- John Reynolds was building a programming language for which he invented $\lambda 2$ to act as a foundational calculi.

- A hard question to answer (and took 25 years to answer by Joe Wells) was: Is type checking decidable in $\lambda 2$?

- I.e., if you give me a non type-annotated term of $\lambda 2$, can I find its type?

- This is extremely important because types contain safe programs and so we need to type our terms to guarantee safety of programs.

- Because the question of decidability of type checking was hard to answer, Robin Milner developed his language ML using a small part of $\lambda 2$ (for which type checking was known to be decidable).

- This meant that ML is not modelled by $\lambda 2$ and hence the properties of $\lambda 2$ cannot be exported to ML.

- A lot of research was done to establish the properties and the power of the programming language ML.

ML

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

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

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

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

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

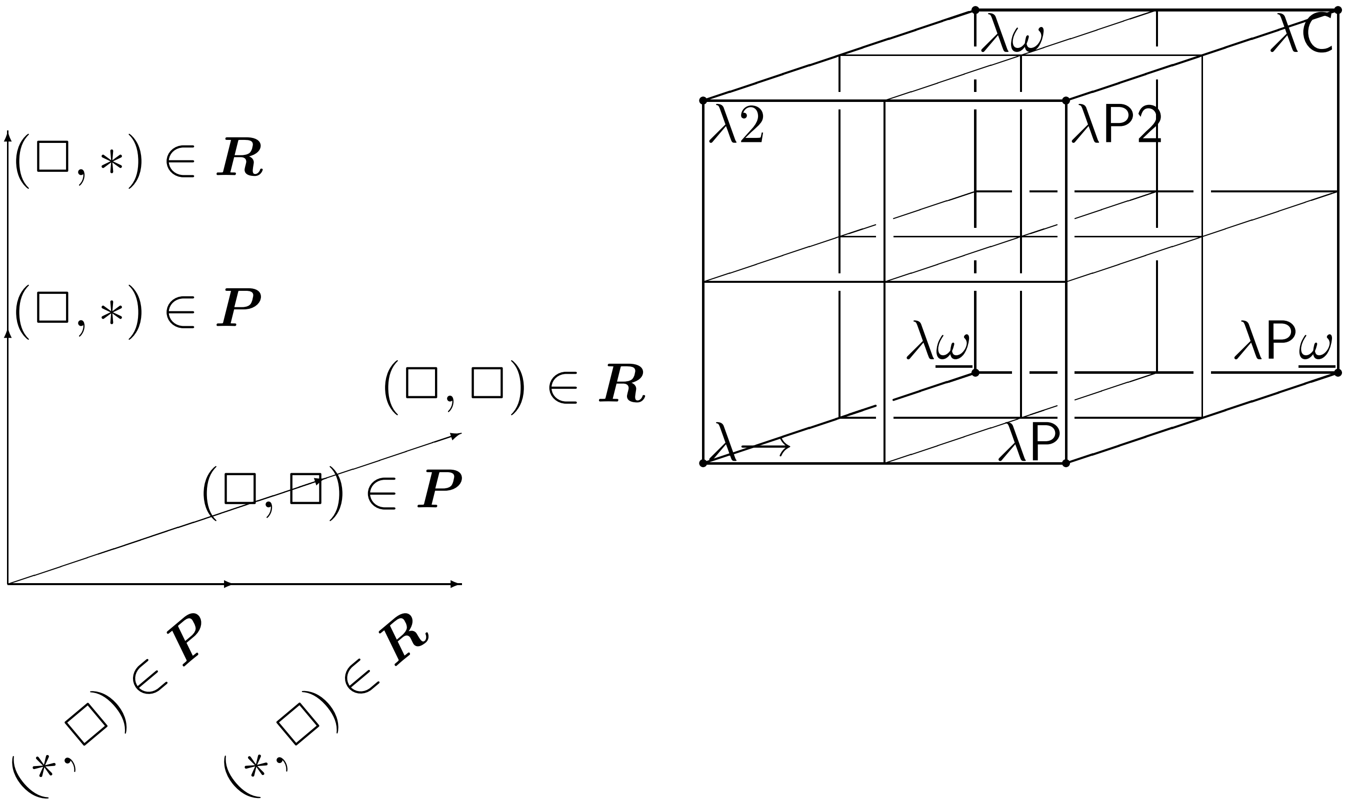
- LF [Harper et al., 1987] is often described as λP of the Barendregt Cube.
- Use of Π-formation rule $(*, \Box)$ is very restricted in the practical use of LF.
- The only need for a type $\Pi x{:}A.B : \Box$ is when the Propositions-As-Types principle PAT is applied during the construction of the type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

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

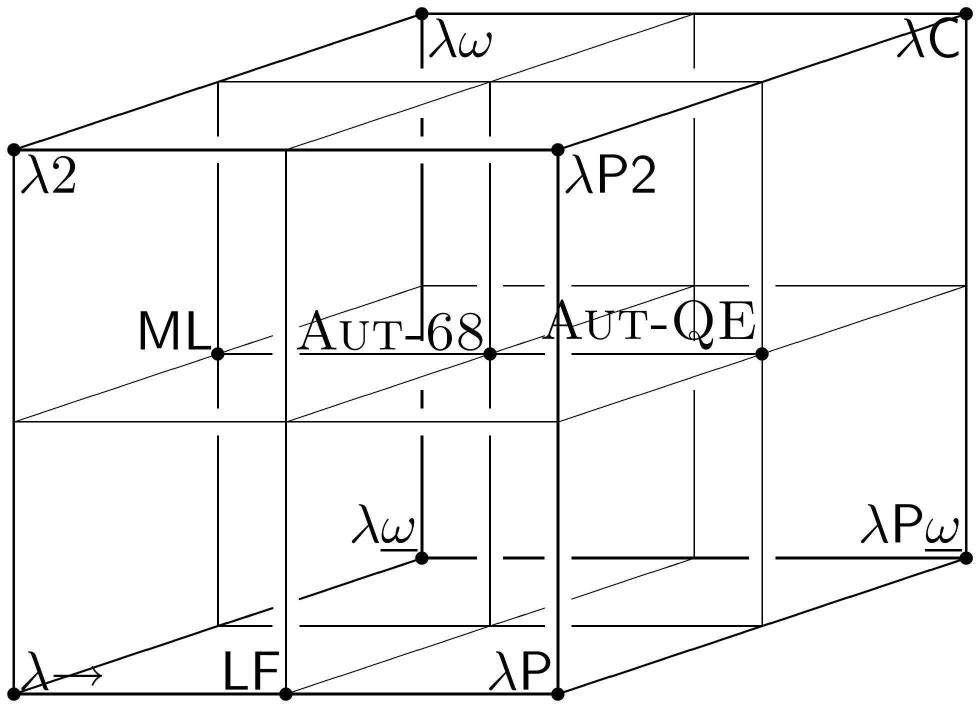
- In LF, this is the only point where the Π-formation rule $(*, \Box)$ is used.
- But, Prf is only used when applied $\Sigma{:}\mathtt{prop}$. We never use Prf on its own.
- This use is in fact based on a parametric constant rather than on Π-formation.
- Hence, the practical use of LF would not be restricted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

A refined version of the cube [Kamareddine et al., 2003]



LF, ML, AUT-68, and AUT-QE in the refined version of the 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.

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 **($\vec{\mathsf{C}}$-weak)** and **($\vec{\mathsf{C}}$-app)**:

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

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

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

Properties of the Refined Cube

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

Example

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

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

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

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

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 $\approx_\beta$.

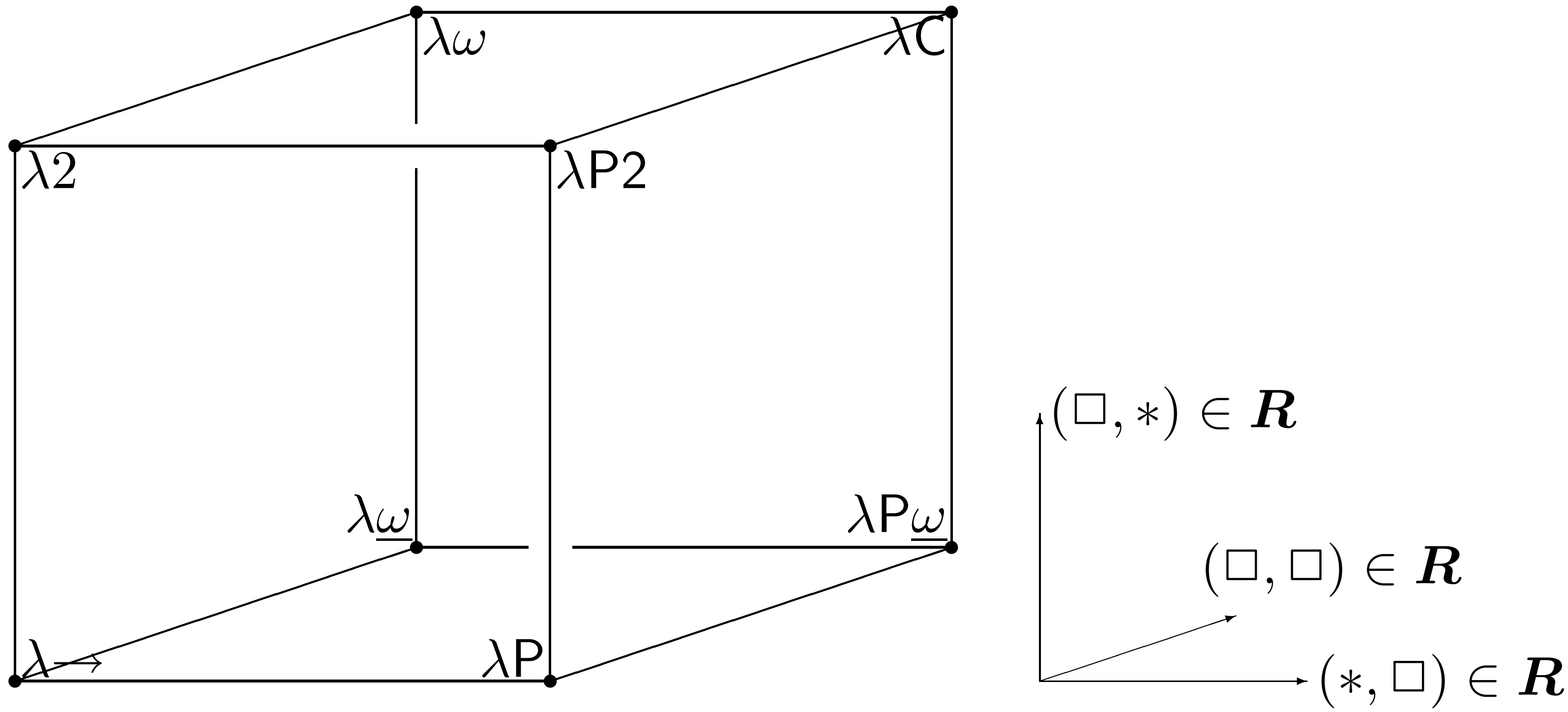


Figure 1: The Barendregt Cube

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

- we define $\Gamma \vdash^c \cdot =_{\texttt{def}} \cdot$ to be the equivalence relation generated by:
  - if $A =_\beta B$ then $\Gamma \vdash^c A =_{\texttt{def}} B$
  - 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 $\Gamma \vdash^c A =_{\texttt{def}} B$.

The (simplified) Cube with definitions and class reduction

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

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

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

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

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

Definitions solve subject reduction

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

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

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

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

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

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

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 $\rightsquigarrow\!\!\!\twoheadrightarrow_\beta$.

De Bruijn Indices [de Bruijn, 1972]

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

Classical λ-calculus with de Bruijn indices

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

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

- $$\begin{array}{l} U_k^i(AB) = U_k^i(A)\, U_k^i(B) \\ U_k^i(\lambda A) = \lambda(U_{k+1}^i(A)) \end{array} \qquad U_k^i(\mathtt{n}) = \begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k\,. \end{cases}$$

- $$\begin{array}{lcl} (A_1A_2)\{\!\{\mathtt{i} \leftarrow B\}\!\} & = & (A_1\{\!\{\mathtt{i} \leftarrow B\}\!\})\,(A_2\{\!\{\mathtt{i} \leftarrow B\}\!\}) \\ (\lambda A)\{\!\{\mathtt{i} \leftarrow B\}\!\} & = & \lambda(A\{\!\{\mathtt{i+1} \leftarrow B\}\!\}) \end{array}$$
  $$\mathtt{n}\{\!\{\mathtt{i} \leftarrow B\}\!\} = \begin{cases} \mathtt{n-1} & \text{if } n > i \\ U_0^i(B) & \text{if } n = i \\ \mathtt{n} & \text{if } n < i\,. \end{cases}$$

- Numerous implementations of proof checkers and programming languages have been based on de Bruijn indices.

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

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

$$
\boxed{
\begin{array}{lrcl}
\sigma\text{-}generation & (\lambda A)\, B & \longrightarrow & A\,\sigma^1\, B \\
\sigma\text{-}\lambda\text{-}transition & (\lambda A)\,\sigma^i B & \longrightarrow & \lambda(A\,\sigma^{i+1}\, B) \\
\sigma\text{-}app\text{-}transition & (A_1\, A_2)\,\sigma^i B & \longrightarrow & (A_1\,\sigma^i B)\,(A_2\,\sigma^i B) \\
\sigma\text{-}destruction & \mathtt{n}\,\sigma^i B & \longrightarrow & \begin{cases} \mathtt{n-1} & \text{if} \quad n > i \\ \varphi_0^i\, B & \text{if} \quad n = i \\ \mathtt{n} & \text{if} \quad n < i \end{cases} \\
\varphi\text{-}\lambda\text{-}transition & \varphi_k^i(\lambda A) & \longrightarrow & \lambda(\varphi_{k+1}^i\, A) \\
\varphi\text{-}app\text{-}transition & \varphi_k^i(A_1\, A_2) & \longrightarrow & (\varphi_k^i\, A_1)\,(\varphi_k^i\, A_2) \\
\varphi\text{-}destruction & \varphi_k^i\,\mathtt{n} & \longrightarrow & \begin{cases} \mathtt{n+i-1} & \text{if} \quad n > k \\ \mathtt{n} & \text{if} \quad n \leq k \end{cases}
\end{array}
}
$$

1. The s-calculus (i.e., λs minus σ-generation) is strongly normalising,

2. The λs-calculus is confluent and simulates (in small steps) β-reduction

3. The λs-calculus preserves strong normalisation PSN.

4. The λs-calculus has a confluent extension with open terms λse.

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

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

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

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

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

$\lambda\sigma_{\Uparrow}$

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

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

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

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

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

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

How is λse obtained from λs?

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

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

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

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

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

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

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

Where did the extra rules come from?

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

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

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

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

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

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

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

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

Computerising Mathematical Texts with MathLang

- MathLang:Project started in 2000 by Fairouz Kamareddine and J.B. Wells.
- Ph.D. students in the MathLang team: Maarek (10/2002-6/2007), Retel (11/2004-06/2008), Lamar (10/2006-now), Zengler (01/2008-12/2008).
- There are two influencing questions:
  1. What is the relationship between logic and mathematics
  2. What is the relationship between computer science and mathematics.
- Question 1 has been slowly brewing for over 2500 years.
- Question 2, is more recent but is unavoidable since automation and computation can provide tremendous services to mathematics.
- There are also extensive opportunities from combining progress in logic and automation/computerisation not only in mathematics but also in other areas: bio-Informatics, chemistry, music, Natural Language, etc.

The birth of computation machines, and limits of computability

- The first half of the 20th century saw a surge of different formalisms and saw the birth of computers (Turing machines, Von Neumann's machine, etc).
- E.g., the discovery of Russell's paradox was the reason for the invention of the first type theory.
- There was a competition between set/type/category theory as a better foundation for mathematics.
- The second half of the 20th century would see a surge of programming languages and softwares for mathematics.

Goals of MathLang: Open borders between mathematics, logic and computation

- Ordinary mathematicians *avoid* formal mathematical logic.
- Ordinary mathematicians *avoid* proof checking (via a computer).
- Ordinary mathematicians *may use* a computer for computation: there are over 1 million people who use Mathematica (including linguists, engineers, etc.).
- Mathematicians may also use other computer forms like Maple, LaTeX, etc.
- But we are not interested in only *libraries* or *computation* or *text editing*.
- We want *freedeom of movement* between mathematics, logic and computation.
- At every stage, we must have *the choice* of the level of formalilty and the depth of computation.

Goals of MathLang

Can we formalise a mathematical text, avoiding as much as possible the ambiguities of natural language, while still guaranteeing the following four goals?

1. The formalised text looks very much like the original mathematical text (and hence the content of the original mathematical text is respected).

2. The formalised text can be fully manipulated and searched in ways that respect its mathematical structure and meaning.

3. Steps can be made to do computation (via computer algebra systems) and proof checking (via proof checkers) on the formalised text.

4. This formalisation of text is not much harder for the ordinary mathematician than LaTeX. *Full formalization down to a foundation of mathematics is not required*, although allowing and supporting this is one goal.

Muito Obrigada

Bibliography

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

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

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

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.-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.

F. Kamareddine and R. Bloo. De Bruijn's syntax and reductional behaviour of lambda terms. *Submitted*, 2002.

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.

F. Kamareddine, T. Laan, and R. P. Nederpelt. Types in logic and mathematics before 1940. *Bulletin of Symbolic Logic*, 8(2):185–245, June 2002.

F. Kamareddine, T. Laan, and R. P. Nederpelt. Revisiting the λ-calculus notion of function. *J. Algebraic & Logic Programming*, 54:65–107, 2003.

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

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

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

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

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

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

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

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

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

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

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

L. Regnier. Une équivalence sur les lambda termes. *Theoretical Computer Science*, 126:281–292, 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.