ULTRA: Useful Logics, Types, Rewriting, and Automation

Fairouz Kamareddine
Heriot-Watt University, Edinburgh

April 2003

Overview

Introduction to ULTRA.

Logics, Types, and Rewriting: What and Why.

Type Free lambda calculus

Simply typed lambda calculus

Polymorphic lambda calculus

A Century of Complexity

	1900	2000
Main way information travels in society:	paper	electric signals, radio
Number of parts in complex machine:	10,000 (locomotive)	1,000,000,000 (CPU)
Worst consequences of single machine failure:	100s die	end of all life?
Likelihood a machine includes a computer:	very low	very high

The Need for Formalism

Because of the increasing interdependency of systems and the faster and more automatic travel of information, failures can have a wide impact. So *correctness* is important.

Modern technological systems are just too complicated for humans to reason about unaided, so *automation* is needed.

Systems have so many possible states that *testing* is often impractical. It seems that *proofs* are needed to cover infinitely many situations.

So *some* kind of formalism is needed to *aid in design* and to *ensure safety*.

What Kind of Formalisms?

reasoning formalism should *at least* be:

Correct: Only correct statements can be "proven".

Adequate: Needed properties in the problem domain can be stated and proved.

Feasible: The resources (money, time) used in stating and proving needed properties must be within practical limits.

What Kind of Formalisms?

suming a minimally acceptable formalism, we would also like it to be:

Efficient: Costs of both the reasoning process *and* the thing being reasoned about should be minimized.

Supportive of reuse: Slight specification changes should not force reproving properties for an entire system. Libraries of pre-proved statements should be well supported.

Elegant: The core of the reasoning formalism should be as simple as possible, to aid in reasoning about the formalism itself.

ULTRA Research Themes

Useful

Logics

Logic is the foundation for rigorous reasoning. There is an ongoing search for better logics and for better methods for verifying the correctness of logics.

Types

Types are a foundation for making logics more flexible without losing correctness and safety. Types are also being used increasingly often for analyzing complex higher-order systems.

Rewriting

Rewriting is using rules of logic, mathematics, or computation in a stepwise manner. Rewriting theory supports reasoning about equivalences between propositions or programs and efficient computation strategies.

and their

Automation

Modern theories of logic, types, and rewriting and the systems to which they are applied have become so complicated that automation is essential.

Applications

Systems of logic, types, and rewriting have applications in the design and implementation of programming languages and theorem provers, in mathematics and in natural language.

Overview

Introduction to ULTRA.

Logics, Types, and Rewriting: What and Why.

Type Free lambda calculus

Simply typed lambda calculus

Polymorphic lambda calculus

Proofs? Logics? What are they?

A proof is the *guarantee* of some statement provided by a rigorous *explanation* stated using some *logic*.

A logic is a formalism for statements and proofs of statements. A logic usually has *axioms* (statements "for free") and *rules* for combining already proven statements to prove more statements.

For example, this is provable in the logic PROP:

$$A,\ B,\ A \to B \to C \vdash C$$

This is not:

$$A,\ B,\ A \to D \to C \nvdash C$$

Why do we believe the explanation of a proof? Because a proved statement is derived step by step from explicit assumptions using a trusted logic.

Logic is an Area of Active Research

New logics are regularly invented for specialized purposes. Known logics may be *too inflexible* for the task. Or they may be *too flexible*, interfering with automated proof search.

Broken logics are regularly invented. A recent example: The 1988 version of the OCL (Object Constraint Language) sublanguage of UML (Unified Modelling Language) had Russell's paradox of a nearly a century earlier! It is still not known if the revised OCL and/or UML is consistent.

There has been an explosion of new logics in the 20th century. How do we know which ones to trust?

What are Types?

Euclid's *Elements* (circa 325 B.C.) begins with:

1. A *point* is that which has no part;
2. A *line* is breadthless length.

   ⋮

5. A *circle* is a plane figure contained by one line such that all the straight lines falling upon it from one point among those lying within the figure are equal to one another.

Although the above seems to merely *define* points, lines, and circles, more importantly it *distinguishes* between them.

This prevents undesired reasoning, like considering whether two points (instead of two lines) are parallel.

Undesired reasoning? Euclid would have said: *impossible* reasoning. When considering whether objects are parallel, intuition implicitly forced Euclid to think about the *type* of the objects. Because intuition does not support parallel points, Euclid does not even *try* such reasoning.

Why Types are Needed for Logic

Mathematical systems have become less intuitive, for several reasons:

– very complex or abstract
– formal
– Something without intuition is using the system: a computer.

Non-intuitive systems are vulnerable to *paradoxes*. The human brain's built-in type machinery can fail to warn against an impossible situation. Reasoning can proceed obtaining results that may be wrong or paradoxical.

Example: Russell [1902] and Frege [1902] showed that Naive Set Theory had a *paradox*. Let S be "the set of all sets which do not contain themselves". Then, *both* of these are provable:

$$S \in S \qquad\qquad S \notin S$$

Russell [1908] Russell began the use of *types* to solve this problem.

A Quick Introduction to Rewriting

e all know how to do *algebra*:

$$
\begin{array}{rll}
 & (a+b)-a & \text{by rule } x+y=y+x \\
= & (b+a)-a & \text{by rule } x-y=x+(-y) \\
= & (b+a)+(-a) & \text{by rule } (x+y)+z=x+(y+z) \\
= & b+(a+(-a)) & \text{by rule } x+(-x)=0 \\
= & b+0 & \text{by rule } x+0=x \\
= & b &
\end{array}
$$

ewriting is the action of replacing a subexpression which is matched by an instance of one le of a rule by the corresponding instance of the other side of the same rule. If you know gebra, you understand the basics of rewriting.

Important Issues in Rewriting

Orientation: Usually, most rules can only be used from left to right as in $x + 0 \to x$. Forward use of the oriented rules represents progress in computation. Unoriented rules usually do trivial work as in $x + y = y + x$.

Termination: It is desirable to show that rewriting halts, i.e., to avoid infinite sequences of the form $P \to P_1 \to P_2 \to \cdots$.

Confluence: It is desirable that the result of rewriting is independent of the order in the rules are used. For example, $1 + 2 + 3$ should rewrite to 6, no matter how we evaluate it.

The invention of computers and computability

Types have *always existed* in mathematics, but not explicited until 1879. Euclid avoided *impossible* situations (e.g., two parallel points) via classes/*types*.

In 19th century, controversies in analysis led to logical *precision*.
(*Cauchy*, *Dedekind*, *Cantor*, *Peano*, *Frege*).

In 1900, *Hilbert* posed an impressive list of difficult questions including his 23rd question:

Given a formula of predicate logic, can we decide whether the formula is true or false?

It took more than 30 years to answer this is impossible: *Turing Machines*, *Goedel's incompleteness* and *Church's λ-calculus*.

f is computable iff f can be computed on a Turing Machine.

f is computable iff f can be definable in the λ-calculus.

Types, Logics, and Rewriting have become the heart of Computer Science.

Higher-Order Rewriting and Logic

Church's λ-calculus provides *higher-order* rewriting, allowing equations like:

$$f(\underline{(\lambda x.\, x + (1/x))5}) = f(5 + \underline{(1/5)}) = f(\underline{5 + 0.2}) = f(5.2)$$

Church [1940b] introduced the simply typed λ-calculus (STLC) and on top of it his Simple Type Theory (CSTT) to provide paradox-free logic. The modern descendant of CSTT is the so-called "higher-order logic" (HOL).

The Convergence of Logics, Types, and Rewriting

Heyting [1934], Kolmogorov [1932], Curry and Feys [1958] (improved by Howard [1980]), and de Bruijn [Nederpelt et al., 1994a] all observed the "*propositions as types*" or "*proofs as terms*" (PAT) correspondence.

In PAT, not only is the λ-calculus embedded in the *propositions* as in HOL, but the structure of *proofs* is also given by another level of λ-terms. λ-terms are viewed as proofs of the propositions represented by their types.

Advantages of PAT include:

- better proof manipulation,
- better independent proof checking,
- the extraction of computer programs from proofs, and
- proving the consistency of the logic via the termination of the rewriting system.

Overview

Introduction to ULTRA.

Logics, Types, and Rewriting: What and Why.

Type Free lambda calculus

Simply typed lambda calculus

Polymorphic lambda calculus

Syntax of type free lambda calculus

`http://www.cee.hw.ac.uk/~ceepl1/cs4project/index.htm`

$\mathcal{V} = \{x, y, z, \dots\}$ is an infinite set of *term variables*. We let $v, v', v'', \cdots$ range over $\mathcal{V}$

$\mathcal{M} ::= \mathcal{V} \mid (\lambda \mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M})$. We let $A, B, C \cdots$ range over $\mathcal{M}$.

Examples $(\lambda x.x)$, $(\lambda x.(xx))$, $(\lambda x.(\lambda y.x))$, $(\lambda x.(\lambda y.(xy)))$, and $((\lambda x.x)(\lambda x.x))$.

This simple language is surprisingly rich. Its richness comes from the freedom to create and apply functions, especially higher order functions to other functions (and even to themselves).

Meaning of Terms

Assume a model $\mathcal{D}$ of the lambda calculus. Let ENV $= \{\sigma \mid \sigma : \mathcal{V} \mapsto \mathcal{D}\}$

Variables The meaning of a variable is determined by what the variable is bound to in the *environment*.

Expressions have variables and variables take values according to environment.

Example, if $\mathcal{V} = \{x, y, z\}$ and if $\mathcal{D}$ contains all natural numbers, then one possible environment might be σ where $\sigma(x) = 1$, $\sigma(y) = 3$ and $\sigma(z) = 1$.

Function application If A and B are λ-expressions, then so is (AB). This expression denotes the result of applying the function denoted by A to the meaning of B.

For example, if A denotes the identity function and B denotes the number 3 then AB denotes identity applied to 3 which is 3.

Abstraction $\lambda v.A$ denotes the function which takes an object a and returns the result of applying the function denoted by A in an environment in which V denotes a.

The semantic function

Let $\sigma(a/v) : \mathcal{V} \mapsto \mathcal{D}$ where
$\sigma(a/v)(v') = \sigma(v')$ if $v \neq v'$ and $\sigma(a/v)(v) = a$

Let $[\,]$: $\mathcal{M} \mapsto \text{ENV} \mapsto D$.

$[v]_\sigma = \sigma(v)$.

$[AB]_\sigma = [A]_\sigma([B]_\sigma)$.

$[(\lambda v.A)]_\sigma = f$ where $f : \mathcal{D} \mapsto \mathcal{D}$ and $f(a) = [A]_{\sigma(a/v)}$.

Example: $[(\lambda x.x)]_\sigma = f$ where $f(a) = [x]_{\sigma(a/x)} = \sigma(a/x)(x) = a$.

Hence, $(\lambda x.x)$ denotes the identity function.

Exercises

Exercise, show that $(\lambda x.(\lambda y.x))$ denotes the function which takes two arguments and returns the first.

Represent the following mathematical functions in the λ-calculus:

1. $f : x \to g$ where $g : y \to x + y$.
2. $f : x \to x + y$ and $g : y \to x + y$.
3. The function f that takes three functions g, h, k and composes them.
4. The function f that takes a function g and iterates it five times.

Describe the functions denoted by $(\lambda x.(\lambda y.(xy)))$, $(\lambda x.(\lambda y.y))$ and $(\lambda x.(\lambda y.x))$.

Notational Conventions

Functional application associates to the left. So ABC denotes $((AB)C)$.

The body of a λ is anything that comes after it. So, instead of $(\lambda v.(A_1 A_2 \ldots A_n))$, we write $\lambda v.A_1 A_2 \ldots A_n$.

A sequence of λ's is compressed to one, so $\lambda xyz.t$ denotes $\lambda x.(\lambda y.(\lambda z.t))$.

a consequence of these notational conventions we get:

Parentheses may be dropped: (AB) and $(\lambda v.A)$ are written AB and $\lambda v.A$.

Application has priority over abstraction: $\lambda x.yz$ means $\lambda x.(yz)$ and not $(\lambda x.y)z$.

Do exercises 2 and 3 of tutorial sheet 1 of the web page of the course.

Free and Bound Variables

Evaluating $(\lambda fx.fx)g$ to $\lambda x.gx$ is perfectly acceptable but evaluating $(\lambda fx.fx)x$ to $\lambda x.xx$ is not.

Check the meaning of these two expressions. $\lambda x.gx$ takes a and applies g to a. $\lambda x.xx$ takes a and applies it to itself.

Also, $(\lambda fx.fx)$ is the same as $(\lambda fy.fy)$ but $(\lambda fx.fx)x$ evaluates to $\lambda x.xx$ and $(\lambda fy.fy)x$ evaluates to $\lambda y.xy$

We define the notions of *free* and *bound* variables which will play an important role in avoiding the problem above. The free x in $(\lambda fx.fx)x$ should remain free in the result.

$$
\begin{array}{lll}
FV(v) & =_{def} & \{v\} \\
FV(\lambda v.A) & =_{def} & FV(A) - \{v\} \\
FV(AB) & =_{def} & FV(A) \cup FV(B)
\end{array}
\qquad
\begin{array}{lll}
BV(v) & =_{def} & \emptyset \\
BV(\lambda v.A) & =_{def} & BV(A) \cup \{v\} \\
BV(AB) & =_{def} & BV(A) \cup BV(B)
\end{array}
$$

Exercise: In $(\lambda y.x(\lambda x.x))$ which variables are bound and which are free?

Substitution

For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:

$$
\begin{array}{lcll}
v[v := B] & \equiv & B & \\
v'[v := B] & \equiv & v & \text{if } v \not\equiv v' \\
(AC)[v := B] & \equiv & A[v := B]C[v := B] & \\
(\lambda v.A)[v := B] & \equiv & \lambda v.A & \\
(\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] & \\
 & & \text{if } v' \not\equiv v \text{ and } (v' \notin FV(B) \text{ or } v \notin FV(A)) & \\
(\lambda v'.A)[v := B] & \equiv & \lambda v''.A[v' := v''][v := B] & \\
 & & \text{if } v' \not\equiv v \text{ and } (v' \in FV(B) \text{ and } v \in FV(A)) &
\end{array}
$$

So, in $(\lambda x.fx)[f := x]$, as $x \in FV(x)$ and $f \in FV(fx)$, we use last clause and get $(\lambda x.fx)[x := y][f := x] = (\lambda y.fy)[f := x] = (\lambda y.xy)$.

Calculate $(\lambda x.y)[y := x]$. Why do we disallow the result to be $\lambda x.x$?

Exercises

Evaluate:

1. $(\lambda x.xy)[x := \lambda z.z]$
2. $(\lambda y.x(\lambda x.x))[x := \lambda y.yx]$
3. $(y(\lambda z.xz))[x := \lambda y.zy]$

Check that:

- $(\lambda y.yx)[x := z] \equiv \lambda y.yz$,
- $(\lambda y.yx)[x := y] \equiv \lambda z.zy$,
- $(\lambda y.yz)[x := \lambda z.z] \equiv \lambda y.yz$.

ALPHA Reduction

Compatibility

$$\frac{A \to B}{AC \to BC} \qquad \frac{A \to B}{CA \to CB} \qquad \frac{A \to B}{\lambda v.A \to \lambda v.B}$$

$\to_\alpha$ is defined to be the least compatible relation closed under the axiom:

$$(\alpha) \qquad \lambda v.A \to_\alpha \lambda v'.A[v := v'] \qquad \text{where } v' \notin FV(A)$$

$\twoheadrightarrow_\alpha$ is the reflexive, transitive closure of $\to_\alpha$.

$=_\alpha$ is the reflexive, transitive, symmetric closure of $\to_\alpha$.

$\lambda x.x \to_\alpha \lambda y.y$ but it is not the case that $\lambda x.xy \to_\alpha \lambda y.yy$.
Moreover, $\lambda z.(\lambda x.x)x \twoheadrightarrow_\alpha \lambda z.(\lambda y.y)x$.

$\lambda x.x =_\alpha \lambda y.y$.

BETA Reduction

$\rightarrow_\beta$ is defined to be the least compatible relation closed under the axiom:

$$(\beta) \qquad (\lambda v.A)B \rightarrow_\beta A[v := B]$$

$\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\rightarrow_\beta$.

$=_\beta$ is the reflexive transitive, symmetric closure of $\rightarrow_\beta$.

We say that A is in β-normal form if there is no B such that $A \rightarrow_\beta B$.

Check that:

- $(\lambda x.x)(\lambda z.z) \rightarrow_\beta \lambda z.z$,
- $(\lambda y.(\lambda x.x)(\lambda z.z))xy \twoheadrightarrow_\beta y$,
- both $\lambda z.z$ and y are β-normal forms.

Exercises

Give the sets of free and bound variables of the following λ-terms and for each variable occurrence, say whether it is bound or free:

1. $\lambda x.\lambda y.(\lambda y.(\lambda z.z)\lambda y.z)y$
2. $(\lambda x.(\lambda y.\lambda z.pq)y)xz$
3. $\lambda x.yz(\lambda yz.y)x$

For each of the above terms apply β-reduction until no β-redexes can be found.

Metatheory

Some Programs loop/don't terminate: $\underline{(\lambda x.xx)(\lambda x.xx)}$ does not have a normal form.

We can evaluate programs in different orders, but always get the same final result:
$(\lambda y.\underline{(\lambda x.x)(\lambda z.z)})xy \to_\beta \underline{(\lambda y.\lambda z.z)xy} \to_\beta \underline{(\lambda z.z)y} \to_\beta y$ and
$\underline{(\lambda y.(\lambda x.x)(\lambda z.z))xy} \to_\beta \underline{((\lambda x.x)(\lambda z.z))y} \to_\beta \underline{(\lambda z.z)y} \to_\beta y$

The order we use to evaluate programs can affect termination:
A term may be normalising but not strongly normalising:
$\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \to_\beta z$ yet
$(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta \ldots$

A *program may grow after reduction:*

$$\begin{array}{lll} \underline{(\lambda x.xxx)(\lambda x.xxx)} & \to_\beta & \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx) \\ & \to_\beta & \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx)(\lambda x.xxx) \\ & \to_\beta & \ldots \end{array}$$

If an expression β-reduces in two different ways to two values, then those values, if they are in β-normal form are the same (up to α-conversion).

$$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda x.x) \to_\beta$$
$$\underline{(\lambda yz.z(yz))(\lambda x.x)} \to_\beta \lambda z.z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.zz.$$

$$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \to_\beta$$
$$\lambda z.\underline{(\lambda x.x)z}((\lambda x.x)z) \to_\beta \lambda z.z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.zz.$$

$$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \to_\beta$$
$$\lambda z.(\lambda x.x)z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.\underline{(\lambda x.x)zz} \to_\beta \lambda z.zz.$$

Church-Rosser Theorem

$$\forall A, B, C \in \mathcal{M} \;\; \exists D \in \mathcal{M} \;\; : (A \twoheadrightarrow_\beta B \wedge A \twoheadrightarrow_\beta C) \Rightarrow (B \twoheadrightarrow_\beta D \wedge C \twoheadrightarrow_\beta D).$$

Call by Name and Call by Value

Normal Order/Call by name: At every stage, reduce the leftmost-outermost redex. E.g., $\underline{(\lambda y.y)((\lambda x.x)1)} \to \underline{(\lambda x.x)1} \to 1$.

Applicative Order/Call by value: At every stage, reduce the leftmost-innermost redex. E.g., $(\lambda y.y)(\underline{(\lambda x.x)1}) \to \underline{(\lambda y.y)1} \to 1$.

If a program terminates, call by name reduction will reach final value but call by value may not.

Call by Name: $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \to_\beta z$ yet

Call by Value: $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta \ldots$

Call by value is faster than call by name:

Call by Value: $(\lambda x.xx)(\underline{(\lambda y.y)(\lambda z.z)}) \to \underline{(\lambda x.xx)(\lambda z.z)} \to \underline{(\lambda z.z)(\lambda z.z)} \to (\lambda z.z)$.

Call by Name: $\underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} \to \underline{((\lambda y.y)(\lambda z.z))}((\lambda y.y)(\lambda z.z)) \to \underline{(\lambda z.z)((\lambda y.y)(\lambda z.z))} \to \underline{(\lambda y.y)(\lambda z.z)} \to (\lambda z.z)$

Booleans in λ-calculus

$$
\begin{array}{lcl}
\text{true} & \equiv & \lambda xy.x \\
\text{false} & \equiv & \lambda xy.y \\
\text{not} & \equiv & \lambda x.x \text{ false } \text{ true} \\
\text{and} & \equiv & \lambda xy.xy \text{ false} \\
\text{or} & \equiv & \lambda xy.x \text{ true } y \\
\text{if } M \text{ then } N_1 \text{ else } N_2 & \equiv & MN_1N_2
\end{array}
$$

$$
\begin{array}{lcl}
\text{and true true} & =_\beta & \text{true true false} \\
& =_\beta & (\lambda xy.x) \text{ true false} \\
& =_\beta & (\lambda y.\ \text{true}) \text{ false} \\
& =_\beta & \text{true}
\end{array}
$$

$$
\begin{array}{lcll}
\text{if true then } N_1 \text{ else } N_2 & =_\beta & \text{true } N_1N_2 & \\
& =_\beta & (\lambda y.N_1)N_2 & \\
& =_\beta & N_1 & \text{Note that } y \notin FV(N_1)
\end{array}
$$

Numerals in λ-calculus

$0 \equiv \lambda xy.y$, $1 \equiv \lambda xy.xy$, $2 \equiv \lambda xy.x(xy)$ and so on.

$$
\begin{array}{lcl}
S & \equiv & \lambda xyz.xy(yz) \\
A & \equiv & \lambda xyzp.xz(yzp) \\
M & \equiv & \lambda xyz.x(yz) \\
E & \equiv & \lambda xy.yx \\
Z & \equiv & \lambda x.x(\mathsf{true}\ \mathsf{false})\mathsf{true}
\end{array}
$$

$$
\begin{array}{lcl}
Sn & =_\beta & n+1 \\
Amn & =_\beta & m+n \\
Z0 & =_\beta & \mathsf{true} \\
Z(Sn) & =_\beta & \mathsf{false}
\end{array}
$$

Recursion in λ-calculus

a is a fixed point of E if $Ea = a$.

Every program E (term of λ-calculus) has a fixed point:

Let $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$ and let $a = (YE)$.

$Ea = a$: because: $a = (YE) = \underline{(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))E} = \underline{(\lambda x.E(xx))(\lambda x.E(xx))} = E((\lambda x.E(xx))(\lambda x.E(xx))) = E(YE) = Ea.$

With the presence of fixed points, we can solve recursive equations;

fact $\equiv \lambda x.$ if Zx then 1 else $Mx($ fact $(Px))$

fact is defined in terms of itself.

Let $E \equiv \lambda yx.$ if Zx then 1 else $Mx(y(Px))$.
As we see, E is defined in terms of things that already exist and not in terms of itself.

Now, we take fact $\equiv (YE)$, and so, as $E(YE) = (YE)$ we have:
fact $= E($fact$) = \lambda x.$ if Zx then 1 else $Mx($ fact $(Px))$.

PAIRING in λ-calculus

$$\begin{array}{lcl} \text{pair} & \equiv & \lambda xyz.zxy \\ \text{fst} & \equiv & \lambda x.x \text{ true} \\ \text{snd} & \equiv & \lambda x.x \text{ false} \end{array}$$

is easy to prove that

fst(pair $AB) = A$

snd(pair $AB) = B$

fst(pair AB) $=$ ($\lambda x.x$ true)(pair AB) $=$ (pair AB) true $=$ pair AB true $=$ $(\lambda xyz.zxy)AB$ true $=$ true $AB = (\lambda xy.x)AB = A$.

snd(pair AB) $=$ ($\lambda x.x$ false)(pair AB) $=$ (pair AB) false $=$ pair AB false $=$ $(\lambda xyz.zxy)AB$ false $=$ false $AB = (\lambda xy.y)AB = B$.

LISTS in λ-calculus

The equation $xy = x$ has a solution $\perp$ where $\perp y = \perp$ for any y.

Le $E = \lambda xy.x$ and let $\perp = YE$. Then $YE = E(YE)$.
So, $\perp = E\perp$ and $\perp y = E\perp y = (\lambda xy.x)\perp y = \perp$.

$$
\begin{array}{lcl}
\text{null} & \equiv & \text{fst} \\
[\,] & \equiv & \text{pair}\ \ \text{true}\ \perp \\
[E] & \equiv & \text{pair}\ \ \text{false}\ (\text{pair}\ E\ [\,]) \\
[E_1, \ldots, E_n] & \equiv & \text{pair}\ \ \text{false}\ (\text{pair}\ E_1\ [E_2, \ldots E_n])
\end{array}
$$

Exercise: null $[\,] =$ true and null $[E_1, \ldots, E_n] =$ false

null $[\,] =$ fst(pair true $\perp$) $=$ true .

null $[E_1, \ldots, E_n] =$ fst(pair false (pair E_1 $[E_2, \ldots E_n]$)) $=$ false

$$
\begin{array}{lcl}
\text{hd} & \equiv & \lambda l.\ \text{if } (\text{null } l) \text{ then } \bot \text{ else } (\text{fst } (\text{snd } l)) \\
\text{tl} & \equiv & \lambda l.\ \text{if } (\text{null } l) \text{ then } \bot \text{ else } (\text{snd } (\text{snd } l)) \\
\text{cons} & \equiv & \lambda x l.\ \text{pair false } (\ \text{pair } x\, l)
\end{array}
$$

ercises:

null $(\text{cons } x\, l) = \text{fst } ((\lambda x l.\ \text{pair false } (\ \text{pair } x\, l))x\, l) = \text{fst } (\ \text{pair false } (\ \text{pair } x\, l)) =$ false

snd $(\text{cons } x\, l) = \text{snd } (\text{pair false } (\ \text{pair } x\, l)) = (\ \text{pair } x\, l)$

hd $(\text{cons } x\, l) = (\lambda l.\ \text{if } (\text{null } l) \text{ then } \bot \text{ else } (\text{fst } (\text{snd } l)))(\text{cons } x\, l) =$
if $(\text{null } (\text{cons } x\, l))$ then $\bot$ else $(\text{fst } (\text{snd } (\text{cons } x\, l))) =$
if false then $\bot$ else $(\text{fst } (\text{snd } (\text{cons } x\, l))) =$
$\text{fst } (\text{snd } (\text{cons } x\, l)) = \text{fst } (\ \text{pair } x\, l) = x$

tl $(\text{cons } x\, l) = (\lambda l.\ \text{if } (\text{null } l) \text{ then } \bot \text{ else } (\text{snd } (\text{snd } l)))(\text{cons } x\, l) =$
if $(\text{null } (\text{cons } x\, l))$ then $\bot$ else $(\text{snd } (\text{snd } (\text{cons } x\, l))) =$
if false then $\bot$ else $(\text{snd } (\text{snd } (\text{cons } x\, l))) =$
$\text{snd } (\text{snd } (\text{cons } x\, l)) = \text{snd } (\ \text{pair } x\, l) = l$

Find a λ-expression *append* such that

$$\text{append } x\, y = \text{ if (null } x) \text{ then } y \text{ else (cons(hd } x)(\text{append (tl } x)y)$$

Let $E = \lambda axy.$ if (null x) then y else (cons (hd x)(a(tl x)y) and let append $= YE$.

Then, append $xy = YExy = E(YE)xy = E(\text{append})xy =$
$(\lambda axy.$ if (null x) then y else (cons (hd x)(a(tl x)y)))append $xy =$
if (null x) then y else (cons (hd x)(append(tl x)y)

UNDECIDABILITY of HALTING

Note that $\perp$ loops: $\perp = YE = E(YE) = E(E(YE)) = E(E(E(YE)))\ldots$.

Let halts $E =$ true if E has a normal form and halts $E =$ false otherwise.

halts is *Not* definable in the λ-calculus.

Assume the contrary (i.e. halts is a λ-term), then

Let foo $= \lambda x.$ if (halts x) then $\perp$ else 0.

Let W be a solution to $x =$ foo x. Hence, $W =$ foo $W =$ if (halts W) then $\perp$ else 0.

Case halts W is true then $W =$ if (halts W) then $\perp$ else $0 = \perp$.
Absurd as $\perp$ does not have normal form.

Case halts W is false then $W =$ if (halts W) then $\perp$ else $0 = 0$.
Absurd as 0 does have a normal form.

Hence, what we assumed is false and so, halts is not a lambda term.

Overview

Introduction to ULTRA.

Logics, Types, and Rewriting: What and Why.

Type Free lambda calculus

Simply typed lambda calculus

Polymorphic lambda calculus

Church's Simply Typed λ-calculus $\lambda\!\!\rightarrow$ 1940

Types • *Basic* individuals/propositions • *Arrows* $\alpha \to \beta$

Examples of types: $(\alpha \to \beta) \to (\beta \to \gamma)$, $\alpha \to (\beta \to \gamma)$, $Bool \to Bool$.

Terms *variables*, AB, $\lambda x{:}\alpha.A$.

(β) $(\lambda x{:}\alpha.A)B \to_\beta A[x := B]$.

Start If $(x : \alpha) \in \Gamma$ then $\Gamma \vdash x : \alpha$.

$\to$-introduction If $\Gamma, x{:}\alpha \vdash A : \beta$ then $\Gamma \vdash \lambda x{:}\alpha.A : \alpha \to \beta$

$\to$-elimintation If $\Gamma \vdash A : \alpha \to \beta$ and $\Gamma \vdash B : \alpha$ then $\Gamma \vdash AB : \beta$

$\lambda x : \alpha.x : \alpha \to \alpha$.

$\lambda x : (\alpha \to \beta).x : (\alpha \to \beta) \to (\alpha \to \beta)$.

$\lambda x : \alpha.\lambda y : \beta.x : \alpha \to (\beta \to \alpha)$.

$$
\begin{array}{lll}
x : \alpha \vdash x : \alpha & & \text{start} \\
\vdash \lambda x : \alpha . x : \alpha \to \alpha & & \to\text{-introduction}
\end{array}
$$

$$
\begin{array}{lll}
x : \alpha \to \beta \vdash x : \alpha \to \beta & & \text{start} \\
\vdash \lambda x : \alpha \to \beta . x : (\alpha \to \beta) \to (\alpha \to \beta) & & \to\text{-introduction}
\end{array}
$$

$$
\begin{array}{lll}
x : \alpha, y : \beta \vdash x : \alpha & & \text{start} \\
x : \alpha \vdash \lambda y : \beta . x : \beta \to \alpha & & \to\text{-introduction} \\
\vdash \lambda x : \alpha . (\lambda y : \beta . x) : \alpha \to (\beta \to \alpha) & & \to\text{-introduction}
\end{array}
$$

But, $\lambda x.xx$ cannot be typed.

Overview

Introduction to ULTRA.

Logics, Types, and Rewriting: What and Why.

Type Free lambda calculus

Simply typed lambda calculus

Polymorphic lambda calculus

In $\lambda\!\!\rightarrow$, the function which takes $f : \mathbb{N} \to \mathbb{N}$ and and $x : \mathbb{N}$ and returns $f(f(x))$ is:

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

and has type

$$(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$

If we want the same function on booleans, we would need to write:

$$\lambda f : \mathcal{B} \to \mathcal{B}.\lambda x : \mathcal{B}.f(f(x))$$

which has type

$$(\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B})$$

Instead of repeating the work, we can write the *Polymorphic* doubling function as:

$$\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x))$$

Now, we can instantiate α to what we need:

$\alpha = \mathbb{N}$ then: $(\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x)))\mathbb{N} = \lambda f : \mathbb{N} \to \mathbb{N}.\lambda x : \mathbb{N}.f(f(x))$.

$\alpha = \mathcal{B}$ then: $(\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x)))\mathcal{B} = \lambda f : \mathcal{B} \to \mathcal{B}.\lambda x : \mathcal{B}.f(f(x))$.

$\alpha = (\mathcal{B} \to \mathcal{B})$ then: $(\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x)))(\mathcal{B} \to \mathcal{B}) =$
$\lambda f : (\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B}).\lambda x : (\mathcal{B} \to \mathcal{B}).f(f(x))$.

So, types can be abstracted over (like for terms) and we can pass types as arguments (like for terms).

But, as we have new terms like $\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x))$, we need to say what their types is.

The type of this function is: $\Pi\alpha : *.(\alpha \to \alpha) \to (\alpha \to \alpha)$.

Common features of modern types and functions

We can *construct* a type by abstraction. (Write $\alpha : *$ for α *is a type*)

- $\lambda y : \alpha.y$, the identity over α *has type* $\alpha \to \alpha$
- $\lambda\alpha : *.\lambda y : \alpha.y$, the polymorphic identity *has type* $\Pi\alpha : *.\alpha \to \alpha$

We can *instantiate* types. E.g., if $\alpha = \mathbb{N}$, then the identity over $\mathbb{N}$

- $(\lambda y : \alpha.y)[\alpha := \mathbb{N}]$ *has type* $(\alpha \to \alpha)[\alpha := \mathbb{N}]$ or $\mathbb{N} \to \mathbb{N}$.
- $(\lambda\alpha : *.\lambda y : \alpha.y)\mathbb{N}$ *has type* $(\Pi\alpha : *.\alpha \to \alpha)\mathbb{N} = (\alpha \to \alpha)[\alpha := \mathbb{N}]$ 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 $\alpha \to \alpha$ as $\Pi y : \alpha.\alpha$ when y not free in α.

Are we getting into self-application/Trouble?

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

The polymorphic identity function can be applied to its type too:
$(\lambda\alpha : *.\lambda y : \alpha.y)(\Pi\alpha : *.\alpha \to \alpha) \to_\beta \lambda y : (\Pi\alpha : *.\alpha \to \alpha).y$

So, we can now apply this result to polymorphic identity:
$(\lambda y : (\Pi\alpha : *.\alpha \to \alpha).y)(\lambda\alpha : *.\lambda y : \alpha.y) \to_\beta (\lambda\alpha : *.\lambda y : \alpha.y)$

Problem??

$$(\lambda\alpha : *.\lambda y : \alpha.y)(\Pi\alpha : *.\alpha \to \alpha)(\lambda\alpha : *.\lambda y : \alpha.y) \to_\beta (\lambda\alpha : *.\lambda y : \alpha.y)$$

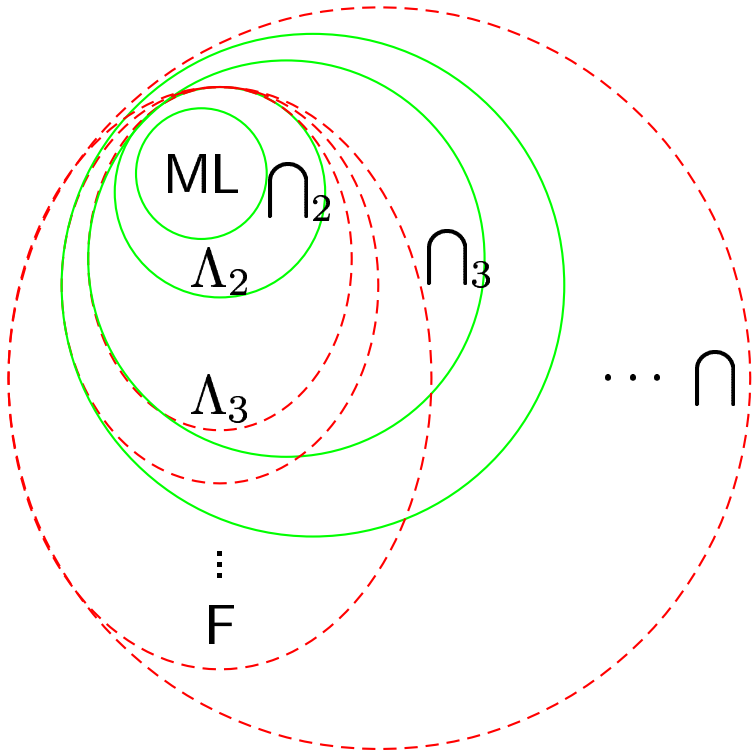
THE NEW SYSTEM IS VERY SAFE.

Subject Reduction: If $\Gamma \vdash A : \alpha$ and $A \to_\beta A'$ then $\Gamma \vdash A' : \alpha$.

Termination: If $\Gamma \vdash A : \alpha$ then both A and α terminate.

More Powerful Type Analysis for Programming Languages

F: System F.
Λ_k: rank-k System F.
$\bigcap$: intersection types.
$\bigcap_k$: rank-k of $\bigcap$.
Decidable.
Undecidable.



. Wells has:

Established the undecidability of System F and its finite-rank restrictions.

Proved decidability of finite-rank restrictions of intersection types at rank 3 and above.

Developed the first understandable analysis algorithms for the systems of intersection types at rank 3 and above.

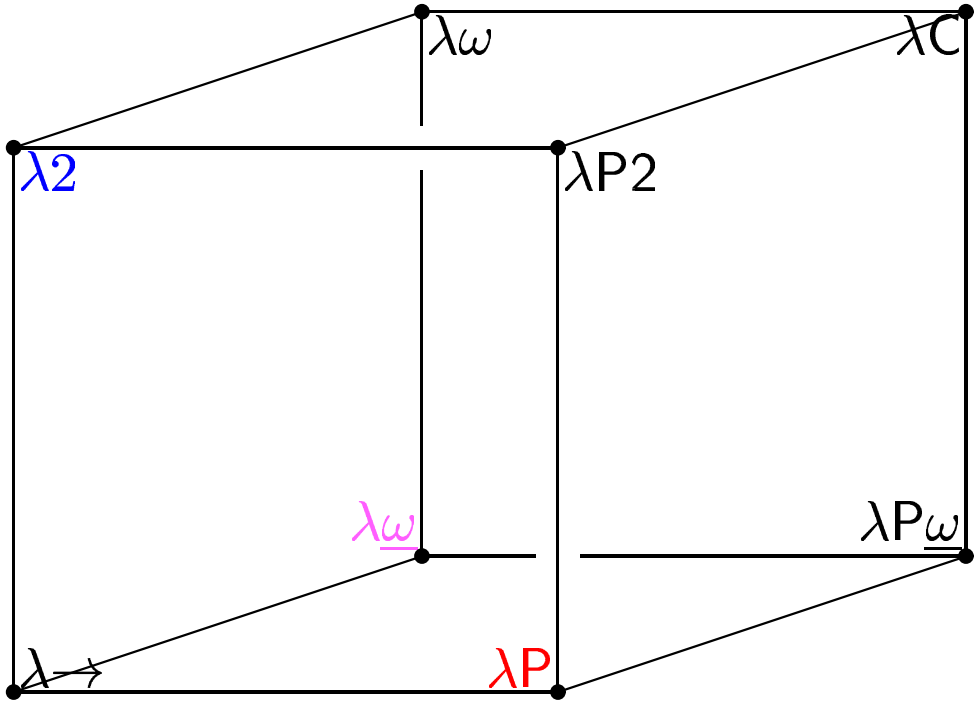
The Barendregt Cube

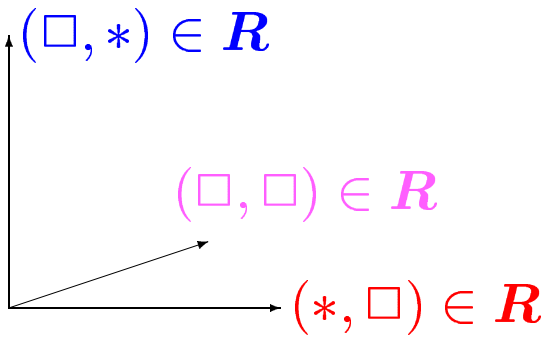
Syntax: $A ::= x \mid * \mid \Box \mid AB \mid \lambda x{:}A.B \mid \Pi x{:}A.B$

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

	Simple	Poly-morphic	Depend-ent	Constr-uctors	Related system	Refs.
$\lambda\!\!\rightarrow$	$(*,*)$				λ^{τ}	Church [1940a]; Barendregt
$\lambda 2$	$(*,*)$	$(\Box,*)$			F	Girard [1972]; Reynolds [197
λP	$(*,*)$		$(*,\Box)$		AUT-QE, LF	Bruijn [1968]; Harper et al.
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$	POLYREC	Renardel de Lavalette [1991
$\lambda P2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$			Longo and Moggi [1988]
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$	Fω	Girard [1972]
$\lambda P\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$		
λC	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$	CC	Coquand and Huet [1988]

The Barendregt Cube





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

$$\frac{y : * \vdash y : * \qquad y : *, x{:}y \vdash y : *}{y : * \vdash \Pi x{:}y.y : *}$$

by (Π) $(*, *)$

$$\frac{y : *, x : y \vdash x : y \qquad y : * \vdash \Pi x{:}y.y : *}{y : * \vdash \lambda x : y.x : \Pi x{:}y.y}$$

by (λ)

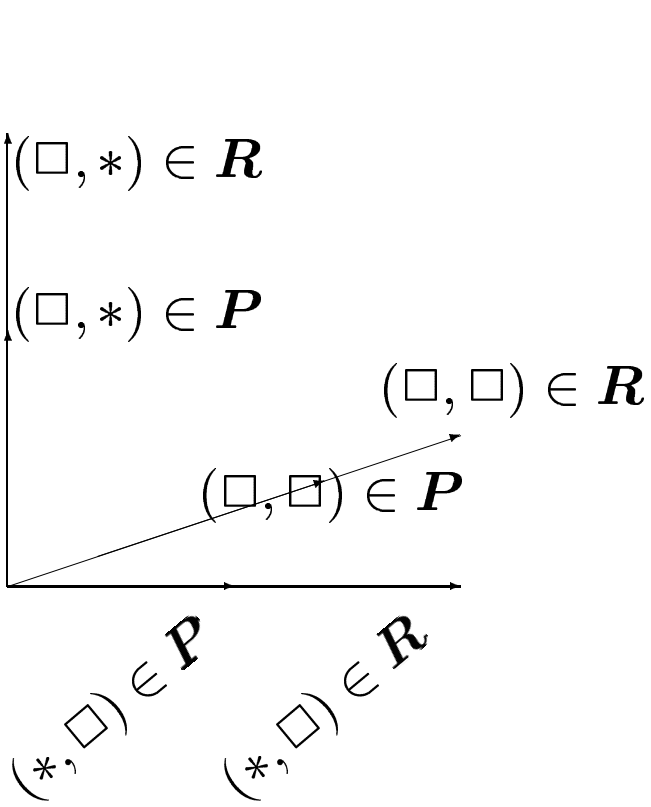
$$\frac{\vdash * : \Box \qquad y : * \vdash \Pi x{:}y.y : *}{\vdash \Pi y : *.\Pi x{:}y.y : *}$$

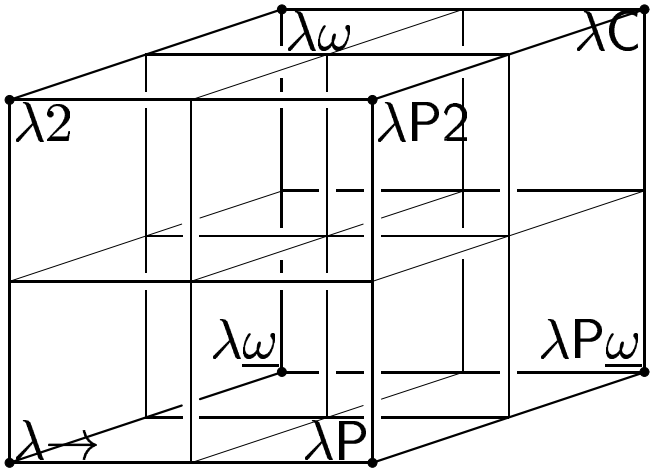
by (Π) $(\Box, *)$

$$\frac{y : * \vdash \lambda x : y.x : \Pi x{:}y.y \qquad \vdash \Pi y : *.\Pi x{:}y.y : *}{\vdash \lambda y : *.\lambda x : y.x : \Pi y : *.\Pi x{:}y.y}$$

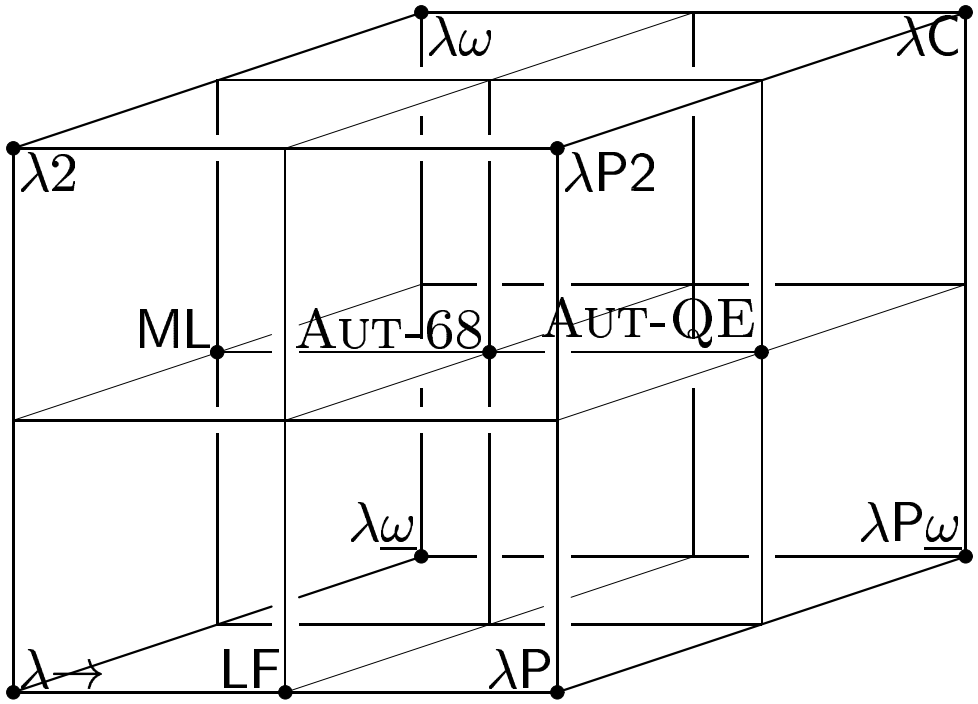
by (λ)

The refined Barendregt Cube





ML in the refined Cube



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.

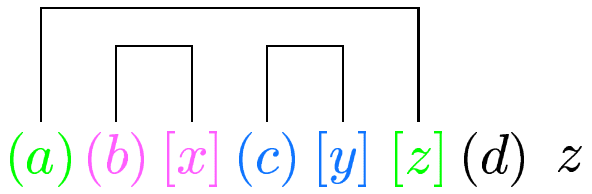
The *applicator wagon* (z) and *abstractor wagon* $[x]$ occur NEXT to each other.

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	$(d)a$



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

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

$\overline{s_i}$ 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:

$$\overline{s} \equiv \overbrace{\emptyset}^{\overline{s_0}}\underbrace{[x][y]}_{\overline{s_1}}\overbrace{(a)[z]}^{\overline{s_2}}\underbrace{[x'](b)}_{\overline{s_3}}\overbrace{(c)(d)[y'][z']}^{\overline{s_4}}\underbrace{(e)}_{\overline{s_5}}$$

References

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

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. [1994b], pages 73–100.

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

onzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940b.

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

B. Curry and R. Feys. *Combinatory Logic I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

Frege. Letter to Russell. English translation in Heijenoort [1967], pages 127–128, 1902.

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

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.

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

Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer-Verlag, Berlin, 1934.

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

. A. Howard. The formulaes-as-types notion of construction. In J. R[oger] Hindley and J[onathan] P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. ISBN 0-12-349050-2. An earlier version was privately circulated in 1969.

N. Kolmogorov. Zur Deutung der Intuitionistischen Logik. *Mathematisches Zeitschrift*, 35:58–65, 1932.

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

ob Nederpelt, J. H. Geuvers, and Roel C. de Vrijer. *Selected Papers on Automath*. North-Holland, Amsterdam, 1994a.

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, 1994b.

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

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

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

Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908. Also in Heijenoort [1967], pages 150–182.