Lambda Calculus and Type Theory
Foundations for Rigorous Software Development

Álvaro Freitas Moreira and Rodrigo Machado

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil
http://www.inf.ufrgs.br

Content

Content

Introduction to Lambda Calculus and Type Theory

Welcome!

Welcome all to

- **Lambda Calculus and Type Theory** (INF05013)
- **Foundations for Rigorous Software Development** (CMP 194)

This discipline will approach the connection between **computation** and **logic**, from both theoretical and practical perspectives.

For each of (lambda calculus, type theory) we start by posing the following questions:

- what is it?
- who created it, and when?
- why it is important?
- why should we bother to know about it **today** (2020)?

Lambda Calculus: what is it?

The (untyped) **Lambda Calculus** was created in the 1930s by Alonzo Church as a formalism that models the essential aspects of function *definition* and *application*.

Church together with his students showed that the lambda calculus can be used for expressing computations (algorithms).

Besides the original version, during the 20th century many **variations** of this formalism were proposed, in particular many distinct **typed lambda calculi**.

Lambda Calculus: who created it, and when?

A brief timeline of lambda calculus and some functional programming languages:

- 1932-1936 : untyped lambda calculus (Church)
- 1940: simply typed lambda calculus (Church)
- 1958: LISP (McCarthy)
- 1966: ISWIN & SECD Machine (Landin)
- 1972-1974: Polymorphic lambda calculus / System F (Reynolds,Girard)
- 1978: Let-polymorphic type inference (Milner)
- 1979: ML (Milner et al.)
- 1985: Calculus of Constructions / Coq (Coquand-Huet)
- 1989: Monads for computations (Moggi)
- 1990: Haskell (Hudak,Hughes,Peyton Jones,Wadler)
- 1991: Lambda cube (Barendregt)
- 2000 onwards: Scala (2001,Odersky), F# (2005, Syme), Idris (2007,Brady), Elixir (2011,Valim), Elm (2012, Czaplicki)

Lambda Calculus: why it is important?

Many important features of the current generation of programming languages came from the study of variations of lambda calculi, and also from the pragmatics of building early functional programming languages.

Examples:

- functions as first-order constructions (LISP)
- managed memory/garbage collection (LISP)
- type inference (ML)
- polymorphism/generics (ML)
- lazy evaluation (SASL, Miranda, Haskell)
- libraries based on monadic combinators (for parsing, testing, etc.) (Haskell)

Lambda Calculus: why should we bother?

From the perspective of programming languages, we expect this discipline will allow you to:

- have a better understanding of anonymous functions in modern programming languages
- project a type system, and also state and prove its properties
- know what it is and how to properly implement and use parametric polymorphism
- know what they are, and how to properly implement and use dependent types
- understand the Curry-Howard isomorphism, and its applications in extending the power of type systems for functional programming languages

Type theory: what is it?

It is a formal system that brings together logic, functional programming, and constructive mathematics.

Central to TT is the duality between **propositions and types**, and between **proofs and programs**:

> a proof of a proposition T seen as a program of type T, and
>
> a program of type T can be seen as a proof of proposition T

For instance, given a proof/program p of proposition/type A, and a proof/program q of proposition/type B we can have a proof/program (a *pair*) (p, q) of proposition $A \wedge B$/type $A \times B$

This duality will be extended for Propositional Logic and also for First Order Logic

Ideas from on side of the divide (Logic or Programming) can be re-interpreted on the other, enriching both fields

Type theory: who created it, and when?

- 1901: Bertrand Russel discovers a paradox in Frege's 1893 *Drundgesetze der Arithmetik* while working on his *Principles of Mathematics* (1903)
- 1902, 1903: Russel writes a letter to inform Frege about the paradox. Russel publish the paradox in *Principles of Mathematics*
- 1908: Russel- Ramified Type Theory to establish a distinction: objects, predicates, predicate of predicates, etc (in *Mathematical Logic as Based on the Theory of Types*
- 1910: Ramified Type Theory used in the three-volume *Principia Mathematica* (an attempt to create a logical basis for mathematics co-authored by Alfred Whitehead)
- 1940: Church's simply typed lambda calculi
- 1967: de Bruijn started implementing Type Theory with Automath
- 1972-1974: Polymorphic lambda calculus (Reynolds)/ System F (Girard)
- 70's: Martin Löf developed intuitionistic/constructive type theories - based on constructive logic and dependent types
- 80's and 90's: Coq proof assistant based on the Calculus of Inductive Constructions - by Thierry Coquand, Gérard Huet and many others

Type theory: why it is important?

Type theory offers a computational and logical language for Computer Science.

Two views of Type Theory for **computer scientists**:

- It brings together logic and programming languages: programming, specification and verification can be done withing a single framework/language (as in Coq, Adga, Lean)

- Impact in functional programming language with features such as type system allowing functions whose type depends on the value of its input, and modules and abstract types whose interfaces can have logical assertions

Type theory: why should we bother ?

Features such as parametric polymorphism (also known as generic types in the object oriented world), and type inference, once considered as academic exercises, are now becoming popular in mainstream languages

Mainstream languages are incorporating many features typical of functional languages

There is a recent trend towards dependent types! Dependent types maybe the next contribution of type theory to make the passage from academia to mainstream languages

Also, knowledge of Type Theory is strongly recommended for those who want to have a deeper understanding of how proof assistants work

Demonstration

Let us present an example of programming and proving in the proof assistant Coq (file “Demonstration in Coq”, available in our Moodle).

Some other cool stuff in Coq:

- Four-color theorem in Coq: `https://github.com/math-comp/fourcolor`
- Compcert C compiler `http://compcert.inria.fr/`

Overview of the content

$$\begin{tikzcd}
\text{[I]PL} \arrow[r, "\text{Quantifiers}"] & \text{[I]FOL}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & \lambda\omega \arrow[rr, "\text{Dependent types}"] & & \lambda C \arrow[r, dashed, "\text{Inductive types}"] & \text{CIC} \\
 & \lambda 2 \arrow[ur, "\text{Type constructors}"] \arrow[rr, dashed] & & \lambda P2 \arrow[ur, dashed] & & \\
 & & \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu, dashed] & \\
\lambda \arrow[r, "\text{Simple types}"'] & \lambda_{\to} \arrow[uu, "\text{Polymorphism}"] \arrow[ur, dashed] \arrow[rr, dashed] & & \lambda P \arrow[uu, dashed] \arrow[ur, dashed] & &
\end{tikzcd}$$

Evaluation and grading

- Four lists of exercises L_1, L_2, L_3, and L_4 to be solved individually
- If $\mathsf{N} \geq 6$, where $\mathsf{N} = (\mathsf{L}_1 + \mathsf{L}_2 + \mathsf{L3} + \mathsf{L}_4)/4$, and class attendance $\geq 75\%$: approved
- If $\mathsf{N} < 6$ and class attendance $\geq 75\%$: exam
- If exam ≥ 6: approved

References

- Rob Nederpelt and Herman Geuvers: **Type Theory and Formal Proof: An Introduction**, Cambridge University Press, 2014. `https://doi.org/10.1017/CBO9781139567725`
- Morten Heine Sorensen and Paweł Urzyczyn: **Lectures on the Curry-Howard Isomorphism**, Elsevier, 2006. Texto completo (acessível via proxy da UFRGS) em `https://bit.ly/2E7Pi8s`
- Henk Barendregt: **Lambda-Calculus with Types**, Cambridge University Press, 2013. Texto completo (acessível via proxy da UFRGS) em `https://bit.ly/2ONmiVI`
- Benjamin Pierce: **Types and Programming Languages**, MIT Press, 2002. Texto completo (acessível via proxy da UFRGS) em `https://bit.ly/3fSXEyA`
- Jean Yves Girard, Yves Lafont and Paul Taylor: **Proofs and Types** (Cambridge University Press, 1989). Texto completo (acessível livremente) em `http://www.paultaylor.eu/stable/Proofs+Types.html`

Content

Lambda Calculus

Function definition and application

Consider the following expression:

$$f(x) = x^2 + 7$$

We think of the equation above as *defining a function* f, which receives a number and generates a new one as result.

$$f(3) = 3^2 + 7 = 16$$

Conversely, we consider the equation above to express that the result of applying f to the specific value 3, which provides the number 16.

Lambda notation solves the following ambiguity:

$$f(e) = e^2 + 7$$

Is the above equation observing the result of f applied to $e = 2.716\ldots$, or is it defining f?

Lambda notation

Lambda notation allows to explicitly distinguish between *function definition* and *function application*.

$$f = \lambda x.x^2 + 7$$

Above we define function f.

- The header λx indicates that x must be interpreted as a *formal parameter*, i.e., a temporary name for the value the function will eventually receive when applied.
- The expression $x^2 + 7$ is the body of the function, and describes what to do with the parameter to obtain the resulting value.

$$f(3) = (\lambda x.x^2 + 7)(3) = 3^2 + 7$$

Above we show the application of function f to 3. Notice that the evaluation of the function application corresponds to substituting the *formal parameter* x by the actual argument 3 within the function body.

Syntax of pre-terms

We begin defining the set Λ^- of **pre-terms**.

Consider an infinite (but countable) set of *names* (a.k.a parameters, identifiers, variables) which we refer by Var.

Let us denote $x, y, z, \ldots$ the elements of Var.

Definition: the set Λ^- is the **smallest set** such that:

1. if $x \in \mathsf{Var}$ then $x \in \Lambda^-$ (variables)
2. if $M \in \Lambda^-$ and $N \in \Lambda^-$ then $@(M, N) \in \Lambda^-$ (application)
3. if $x \in \mathsf{Var}$ and $M \in \Lambda^-$ then $\lambda x.M \in \Lambda^-$ (abstraction)

Notation: we use simply $M\ N$ to denote the application $@(M, N)$.

Examples of pre-terms

Some examples of pre-terms (elements of Λ^-):

x	$\lambda x.y$	$(\lambda x.x)\ (\lambda x.x)$
$x\ y$	$(\lambda x.x)\ y$	$\lambda x.\lambda y.x$
$\lambda x.x$	$\lambda x.(x\ y)$	$\lambda x.\lambda y.y\ (\lambda x.y)$

Notice: notation rules are required to avoid ambiguity when writing pre-terms.

Example: does $\lambda x.x\ y$ represent $\lambda x.(x\ y)$ or $(\lambda x.x)\ y$?

Notation rules

1. The header $\lambda x.$ contains everything to its right as the body (until explicitly stopped by parentheses).

$$\lambda x.x\ y = \lambda x.(x\ y) \neq (\lambda x.x)\ y$$

2. The @ operator is left-associative.

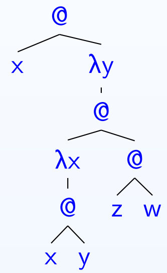
$$x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$$

3. Many lambda abstractions in sequence can be abbreviated.

$$\lambda x\ y\ z.M = \lambda x.\lambda y.\lambda z.M$$

$$x\ \lambda y.(\lambda x.x\ y)(z\ w)$$

$$\Updownarrow$$



Some famous pre-terms

Combinators (from combinatorial logic):

$$\begin{aligned}
\mathbf{I} &= \lambda \mathsf{x}.\mathsf{x} \\
\mathbf{K} &= \lambda \mathsf{x}\ \mathsf{y}.\mathsf{x} \\
\mathbf{S} &= \lambda \mathsf{x}\ \mathsf{y}\ \mathsf{z}.\mathsf{x}\ \mathsf{z}\ (\mathsf{y}\ \mathsf{z})
\end{aligned}$$

Self-application and fixed-point operator:

$$\begin{aligned}
\boldsymbol{\omega} &= \lambda \mathsf{x}.\mathsf{x}\ \mathsf{x} \\
\boldsymbol{\Omega} &= \boldsymbol{\omega}\ \boldsymbol{\omega} \\
\mathbf{Y} &= \lambda \mathsf{f}.\ (\lambda \mathsf{x}.\ \mathsf{f}(\mathsf{x}\ \mathsf{x}))\ (\lambda \mathsf{x}.\ \mathsf{f}(\mathsf{x}\ \mathsf{x}))
\end{aligned}$$

Bound and free variables

Consider the pre-term $\lambda x.M$: we say that M is the **scope** of the header λx.

Every ocurrence of x within the scope of λx is said to be **bound**. If x occurs out of any $\lambda x.$ scope, we say it is **free**.

Example:	a)	$\lambda x.x\ y$	x is bound, y is free
	b)	$\lambda x.z\ \lambda z.z\ x$	z is free, z e x are bound
	c)	$\lambda x.x\ \lambda x.x\ y$	x and x are bound, y is free

Notice:

1. The same variable x can have free and bound occurrences within a given pre-term (b).
2. When x occurs within the scope of more than one $\lambda x.$, it is bound to the innermos header (c).

Bound and free variables (2)

Bound and free occurrences of variables have distinct meanings:

- free variables refer to (external) global names.
- bound variables refer to (local) formal parameters.

Names of *free variables* are important!

$$\text{distinct expressions} \begin{cases} \sin(\pi) - 42 + \pi^2 \\ \sin(e) - 42 + e^2 \end{cases}$$

Names of *bound variables* are not important: they can be exchanged!

$$\text{same definition} \begin{cases} \lambda x.\, \sin(x) - 42 + x^2 \\ \lambda e.\, \sin(e) - 42 + e^2 \end{cases}$$

Bound and free variables (3)

The FV function computes all variable names that occur free in a given pre-term.

$$\begin{array}{ll} \mathsf{FV} : & \Lambda^{-} \to \mathcal{P}(\mathsf{Var}) \\ \mathsf{FV}(\mathsf{x}) = & \{\mathsf{x}\} \\ \mathsf{FV}(\lambda \mathsf{x}.\mathsf{M}) = & \mathsf{FV}(\mathsf{M}) - \{\mathsf{x}\} \\ \mathsf{FV}(\mathsf{M}\ \mathsf{N}) = & \mathsf{FV}(\mathsf{M}) \cup \mathsf{FV}(\mathsf{N}) \end{array}$$

A pre-term M where $\mathsf{FV}(\mathsf{M}) = \{\}$ is called a **closed pre-term** or **combinator**. Otherwise, it is an **open pre-term**.

Substitution

The **substitution operation** $[N/x]M$ substitutes all **free occurrences** of x in M by N.

Example:

1)	$[w/x](\lambda x.x\ y)$	$=$	$\lambda x.x\ y$
2)	$[w/y](\lambda x.x\ y)$	$=$	$\lambda x.x\ w$
3)	$[w/z](\lambda x.x\ y)$	$=$	$\lambda x.x\ y$
4)	$[w/z](z\ \lambda z.z)$	$=$	$w\ \lambda z.z$
5)	$[\lambda z.z\ z/z](z\ z)$	$=$	$(\lambda z.z\ z)(\lambda z.z\ z)$
6)	$[x/y](\lambda x.x\ y)$	$=$	$(\lambda x.x\ x)$?

Question: is there something weird about substitution 6?

Capture of free variables

There is a problem in Ex. 6, called **capture of free variable**.

$$[\overbrace{x}^{N}/y]\,\overbrace{(\lambda x.x\ y)}^{M}$$

- Consider M: in the position of y, the name x is bound.
- Considere N: in this term, the name x is free.
- When y is substituted by N, we put a free occurrence of x in a position where the same name x is bound.
- Since free and bound names have distinct interpretation, transforming a free name into a bound name alters the meaning of the pre-term, which is considered an **error**.
- In the formal definition of substitution, we need to avoid capture of free variables.

Substitution operation: formal definition

Substitution operation avoiding capture of free variables

$$[\_/\_]\_ : \Lambda^- \times \mathsf{Var} \times \Lambda^- \to \Lambda^-$$

$$[P/y]x \;=\; \begin{cases} P & \text{if } x = y \\ x & \text{if } x \neq y \end{cases}$$

$$[P/y](\lambda x.M) \;=\; \begin{cases} \lambda x.M & \text{if } x = y \\ \lambda x.([P/y]M) & \text{if } x \neq y \text{ and } x \notin FV(P) \\ \lambda z.([P/y]([z/x]M)) & \text{if } x \neq y, x \in FV(P) \text{ and} \\ & z \notin \{x, y\} \cup FV(M) \cup FV(P) \end{cases}$$

$$[P/y](M\;N) \;=\; ([P/y]M)\;\;([P/y]N)$$

Alpha equivalence

Question: according to the previous definition of substitution, what is the result of $[x/y](\lambda x.x\ y)$?

Answer: $(\lambda z.z\ x)$ or $(\lambda f.f\ x)$ or $(\lambda m.m\ x)$ …

Two pre-terms M and N are α-equivalent ($M =_\alpha N$) iff they are equal or **only differ** by the choice of names for bound names.

Example:

$$\lambda x.x\ y =_\alpha \lambda z.z\ y \qquad \lambda x.x\ y \neq_\alpha \lambda y.y\ y$$
$$\lambda x.x\ x =_\alpha \lambda z.z\ z \qquad \lambda y.z\ y \neq_\alpha \lambda y.x\ y$$

α-equivalence captures the intuition that the choice of names for bound variables is not important, as long as it does not capture any free variable of the term.

Alpha equivalence: formal definition

We can inductively define $=_\alpha$ as the smallest **relation** over Λ^- such that

$$\frac{}{x =_\alpha x}\ (\alpha\textsc{Var}) \qquad \frac{y \notin FV(M)}{\lambda x.M =_\alpha \lambda y.[y/x]M}\ (\alpha)$$

$$\frac{M =_\alpha M'}{\lambda x.M =_\alpha \lambda x.M'}\ (\alpha\textsc{Lam}) \qquad \frac{M =_\alpha M'}{M' =_\alpha M}\ (\alpha\textsc{Sim})$$

$$\frac{M =_\alpha M' \qquad N =_\alpha N'}{(M\ N) =_\alpha (M'\ N')}\ (\alpha\textsc{App}) \qquad \frac{M =_\alpha M' \qquad M =_\alpha M''}{M =_\alpha M''}\ (\alpha\textsc{Trans})$$

Lemma: for all $M \in \Lambda^-$, $M =_\alpha M$.Proof: by induction on M.

Lemma: if $M =_\alpha M'$, then $FV(M) = FV(M')$. Proof: by induction on $=_\alpha$.

Intuition: all α-equivalent pre-terms have the same set of free variables.

Lambda terms

A **lambda term** is an *equivalence class* of α-equivalent pre-terms.

Notation: denote by $[M]_\alpha$ the lambda term containing pre-term M.

Example:

$$\begin{array}{lcl} [x]_\alpha & = & \{\, x \,\} \\ [\lambda x.x]_\alpha & = & \{\, \lambda x.x,\ \lambda y.y,\ \lambda z.z, \ldots \} \\ [\lambda x.x\ y]_\alpha & = & \{\, \lambda a.a\ y,\ \lambda b.b\ y, \ldots \,\} \end{array}$$

Definition: Λ is the set of all lambda terms.

$$\Lambda = \Lambda^{-}/{=_\alpha}$$

Lambda terms: substitution

Lemma: if $[N/x]M = P$ and $[N/x]M = P'$, then $P =_\alpha P'$.

Proof: by structural induction in M.

The property above allows extending the substitution of Λ^- towards Λ.

Let $[N]_\alpha$ and $[M]_\alpha$ be terms containing, respectively, pre-terms N and M. Define

$$[\_/\_]\_ : \Lambda \times \mathsf{Var} \times \Lambda \to \Lambda$$

$$[[N]_\alpha/x][M]_\alpha = [\ [N/x]M\ ]_\alpha$$

Notice that $[N/x]M$ refers to substitution on pre-terms.

Lambda terms: distribution of substitution

Lemma: if $x \notin FV(P)$, then

$$[P/y]\,([N/x]M) = [[P/y]N/x]\,([P/y]M)$$

Proof: by structural induction on M.

Terms vs pre-terms

From this point onwards, we only refer to terms (and not pre-terms).

However, we will write terms by means of an arbitrary pre-term contained in it: we write “the term M” instead of $[M]_\alpha$.

This is called *Barendregt convention*, and aims to avoid excessive clutter in definitions and proofs.

Notice: equality of terms is actually α-equivalence of pre-terms.

Redexes and normal forms

A **redex** *(reducible expression)* is a subterm of M with the format

$$(\lambda x.P)\ Q$$

and its respective **contractum** is

$$[Q/x]P$$

A term which does not contain any redex is called a **normal form** (or irreducible term).

Example:

a)	$\lambda y.(\lambda x.y\ x)\ ((\lambda y.x)\ y)$	has two redexes
b)	$\lambda x.\lambda y.x\ x$	is a normal form

Beta reduction

Beta reduction describes the evaluation of terms.

We say that M β-reduces to N, denoted $\mathsf{M} \to_\beta \mathsf{N}$, when N is obtained by substituting some redex in M by its respective contractum.

Example:

$$(\lambda z.\lambda x.x\ z)\ y \ \to_\beta\ \lambda x.x\ y$$

$$\lambda y.(\lambda x.y\ x)\ ((\lambda y.x)\ y) \ \to_\beta\ \lambda y.(\lambda x.y\ x)\ x$$

$$\lambda y.(\lambda x.y\ x)\ x \ \to_\beta\ \lambda y.y\ x$$

Beta reduction: formal definition

Definition: $\rightarrow_\beta$ is the least relation on Λ such that

$$(\lambda x.P)\ Q \rightarrow_\beta [Q/x]P \qquad (\beta) \qquad\qquad \frac{M \rightarrow_\beta M'}{N\ M \rightarrow_\beta N\ M'} \qquad (\textsc{app}1)$$

$$\frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'} \qquad (\textsc{abs}) \qquad\qquad \frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N} \qquad (\textsc{app}2)$$

Definition: $\twoheadrightarrow_\beta$ (or $\overset{*}{\rightarrow}_\beta$) is the reflexive and transitive closure of $\rightarrow_\beta$.

Intuitively, $M \twoheadrightarrow N$ when M reduces to N in zero or more beta reduction steps.

Beta equivalence

β-**equivalence** identifies terms with confluente evaluation.

Definition: $=_\beta$ is the smallest *equivalence relation* on Λ such that

$$\frac{M \twoheadrightarrow_\beta P \qquad N \twoheadrightarrow_\beta P}{M =_\beta N}$$

Example: all the following terms are β-equivalent:

$$\begin{aligned} M &= (\lambda z.\lambda x.x\ z)\ y \\ N &= (\lambda u.\lambda x.x\ y)\ (t\ t) \\ P &= \lambda x.x\ y \end{aligned}$$

Notice: notice that $M \not\twoheadrightarrow_\beta N$ and $N \not\twoheadrightarrow_\beta M$.

Terms with normal forms

We say that M **has normal form** iff

$$\mathsf{M} =_\beta \mathsf{N}$$

and N **is normal form**.

Example:

$(\lambda x.x\ x)$ has normal form (it **is** a normal form).

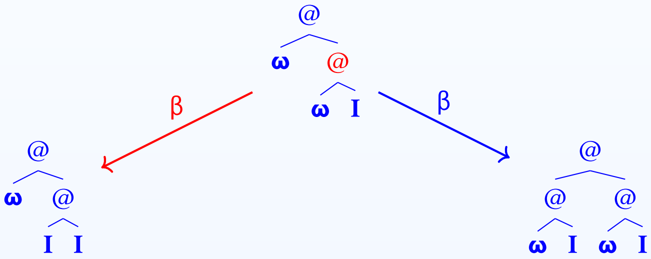
$(\lambda x.x\ x)\ a$ has normal form (it β-reduces to $(a\ a)$, which **is** normal form)

$(\lambda x.x\ x)\ (\lambda x.x\ x)$ does not have normal form (it β-reduces to itself)

Distinct reductions

A term P may have many redexes, and therefore P may β-reduce to distinct terms (depending of the chosen redex).

Example: (consider $\omega = \lambda x.x\ x$ and $\mathbf{I} = \lambda x.x$)



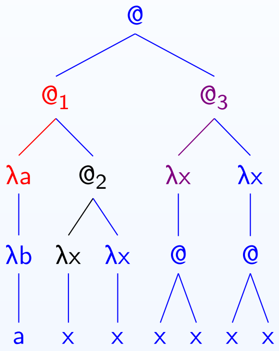
Comparing redexes

Definition: in a term of the format $(\lambda x.P)\ Q$, any redex in P or in Q is considered to be **internal to** the redex $(\lambda x.P)\ Q$ (which is, in turn, considered **external to** the former one).

Definition: in a term of the format $P\ Q$, any redex in P is considered to be **on the left** of any other redex in Q (which is, in turn, considered to be **on the right** of the former one).

Comparing redexes: example

Example: $\mathbf{K}\ (\mathbf{I}\ \mathbf{I})\ \mathbf{\Omega}$, with $\mathbf{K} = \lambda a\ b.a$, $\mathbf{I} = \lambda x.x$ and $\mathbf{\Omega} = (\lambda x.x\ x)\ (\lambda x.x\ x)$:

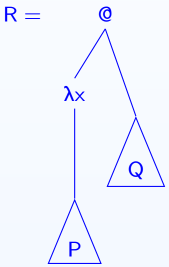


Notice that

- $@_1$ is on the left of $@_3$, and, conversely, $@_3$ is on the right of $@_1$
- $@_2$ is on the left of $@_3$ and, conversely, $@_3$ is on the right of $@_2$
- $@_1$ is external to $@_2$ and, conversely, $@_2$ is internal to $@_1$

Evolution of redexes under beta reduction

Consider a redex $R = (\lambda x.P)\ Q$ within another term M. There may be other redexes in P and Q (internal to R). Assume M' is the result of exchanging R by its contractum $[Q/x]P$ in M. The redexes in M' are as follows:



- the redex R is **deleted** (it does not occur in M')
- an arbitrary redex in Q occurs zero, one or more times in $[Q/x]P$, depending on the number of free occurrences of x in P (notice the **multiplicative** effect).
- any redex within P or outside of R is **preserved** and also occurs in M'.
- **new redexes** may appear:
  - (within $[Q/x]P$) when we have an application $(x\ M)$ in P, and Q is a lambda abstraction;
  - (within M') when R is at the left-hand side of an application within M', and P is a lambda abstraction.

Evaluation strategies

An **evaluation strategy** is a choice of redex (for evaluation).

Normal evaluation: ($\rightarrow_n$)

- leftmost, outermost redex
- beta-reduces without normalizing function and arguments

Applicative evaluation: ($\rightarrow_a$)

- leftmost, innermost redex
- normalizes the function and the argument before beta-reducing

Example:

- normal: $((\lambda x.\lambda y.x)\ \mathbf{I}\ \mathbf{\Omega}) \rightarrow_n (\lambda y.\mathbf{I})\ \mathbf{\Omega} \rightarrow_n \mathbf{I}$
- applicative: $((\lambda x.\lambda y.x)\ \mathbf{I}\ \mathbf{\Omega}) \rightarrow_a (\lambda y.\mathbf{I})\ \mathbf{\Omega} \rightarrow_a (\lambda y.\mathbf{I})\ \mathbf{\Omega} \rightarrow_a \ldots$

Lambda calculus as a model of computation

The *language of lambda terms* together with beta reduction (using the normal evaluation strategy) can be seen as a **model of computation**:

- lambda term = machine state
- beta reduction (normal strategy) = machine execution step
- normal form = halting state

In particular, lambda calculus is a **Turing-complete** model, i.e. it has the same expressive power as Turing machines.

We will employ the tool available via the link below to study the computational power of the lambda calculus.

`http://www.inf.ufrgs.br/~rma/simuladores/lambda.html`

Encodings

There are nothing but variables, function abstraction and function application in the pure lambda calculus.

Therefore, all other elements present in programming languages must be **encoded** using only the former:

Data:

- boolean values and logical operations
- natural numbers, arithmetic and relational operations
- data structures: pairs and lists

Control structures:

- conditional evaluation (if-then-else)
- repetition (recursion or iteration)

Notice: in these encodings we will not worry too much about efficiency.

Boolean values and conditional expressions

Definition: Church booleans:

$$\textbf{true} = \lambda x\ y\ .\ x$$
$$\textbf{false} = \lambda x\ y\ .\ y$$

Intuition:

- **true** receives two arguments and returns the first one
- **false** receives two arguments and returns the second one

Definition: `if-then-else` construct:

$$\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c = \lambda a\ b\ c\ .\ a\ b\ c$$

Idea: it applies the Church boolean over the two possible expressions.

Exercise: define the booleans operations **not**, **and** and **or**.

Natural numbers

Definition: Church numerals:

$$\begin{aligned}
\mathbf{0} &= \lambda f\, x\, .\, x \\
\mathbf{1} &= \lambda f\, x\, .\, f\, x \\
\mathbf{2} &= \lambda f\, x\, .\, f\, (f\, x) \\
\mathbf{3} &= \lambda f\, x\, .\, f\, (f\, (f\, x)) \\
&\vdots \\
\mathbf{n} &= \lambda f\, x\, .\, \overbrace{f\, (f\, (f\, \ldots (f}^{n}\, x) \ldots))
\end{aligned}$$

They will be also referred as $\mathbf{c}_n$, for $n \in \mathbb{N}$.

Arithmetic operations: successor

Successor function:

$$\mathbf{succ} = \lambda \mathsf{n}\ \mathsf{p}\ \mathsf{q}\ .\ \mathsf{p}\ (\mathsf{n}\ \mathsf{p}\ \mathsf{q})$$

Idea:

1. it receives a Church numeral n;

$$\mathsf{n} = \lambda \mathsf{f}\ \mathsf{x}\ .\ \mathsf{f}\ (\mathsf{f} \ldots (\mathsf{f}\ \mathsf{x}) \ldots)$$

2. the returned expression must have p and q as bound variables;

$$\lambda \mathsf{p}\ \mathsf{q}\ .\ \ldots$$

3. the application $(\mathsf{n}\ \mathsf{p}\ \mathsf{q})$ substitutes f's into p's, and x into q;
4. an additional p is applied over $(\mathsf{n}\ \mathsf{p}\ \mathsf{q})$.

Arithmetic and relational operations

Definition: basic arithmetic:

$$\begin{aligned}
\textbf{add} &= \lambda\ m\ n\ p\ q\ .\ m\ p\ (n\ p\ q)\\
\textbf{mult} &= \lambda\ m\ n\ p\ q\ .\ m\ (n\ p)\ q\\
\textbf{exp} &= \lambda\ m\ n\ .\ n\ m
\end{aligned}$$

Definition: testing zero:

$$\textbf{isZero} = \lambda n\ .\ n\ (\lambda x.\textbf{false})\ \textbf{true}$$

Notice: the predecessor function will be introduced after ordered pairs.

Ordered pairs

Definition: the ordered pair (M, N) is represented by $\mathbf{pair}\ \mathsf{M}\ \mathsf{N}$ where

$$\mathbf{pair} = \lambda\, \mathsf{m}\ \mathsf{n}\ \mathsf{b}.\ \mathsf{b}\ \mathsf{m}\ \mathsf{n}$$

Example:

$$\mathbf{pair}\ \mathbf{0}\ \mathbf{true} = \lambda \mathsf{b}.\ \mathsf{b}\ \mathbf{0}\ \mathbf{true}$$

Definition: functions that extract the components of a pair:

$$\mathbf{fst} = \lambda \mathsf{p}.\ \mathsf{p}\ \mathbf{true}$$
$$\mathbf{snd} = \lambda \mathsf{p}.\ \mathsf{p}\ \mathbf{false}$$

Notation: we will also use $\langle \mathsf{M}, \mathsf{N} \rangle$ as a synonym for $(\mathbf{pair}\ \mathsf{M}\ \mathsf{N})$.

Predecessor

Consider the function $\mathsf{pred} : \mathbb{N} \to \mathbb{N}$ below

$$\mathsf{pred}(\mathsf{n}) = \begin{cases} 0 & \text{if} \quad \mathsf{n} \leq 1 \\ \mathsf{n} - 1 & \text{otherwise} \end{cases}$$

Using pairs, we can define the auxiliary function: $\mathsf{shiftInc}(\mathsf{a}, \mathsf{b}) = (\mathsf{b}, \mathsf{b} + 1)$

$$\mathbf{shiftInc} = \lambda \mathsf{p}.\langle \mathbf{snd}\ \mathsf{p},\ \mathbf{succ}\ (\mathbf{snd}\ \mathsf{p})\rangle$$

Definition: **pred** is implemented by the following lambda term pred

$$\mathbf{pred} = \lambda \mathsf{n}.\mathbf{fst}\ (\mathsf{n}\ \mathbf{shiftInc}\ \langle \mathbf{0}, \mathbf{0}\rangle)$$

Lists

Lists are amongst the most used data structures in functional programming languages.

The algebraic definition of lists contains two constructors:

- `empty`, (without arguments) representing the empty list;
- `cons`, which receives a value `v` and a list `l`, and represents the list starting with `v` and whose continuation is `l`.

Example:

```
empty
(cons 2 empty)
(cons 2 (cons 5 (cons 1 empty)))
```

Lists (2)

Three functions access the list structure:

- `isEmpty` tests if the list is empty;
- `head` returns the first element of the list;
- `tail` returns the argument list without its first element.

We consider applying `head` or `tail` over `empty` as an error (with undefined results).

Example:

- `isEmpty empty => true`
- `isEmpty (cons 2 empty) => false`
- `head (cons 2 empty) => 2`
- `tail (cons 3 (cons 2 empty)) => (cons 2 empty)`
- `head (tail (cons 3 (cons 2 empty))) => 2`

Lists (3)

Definition: list constructors:

$$\begin{aligned} \textbf{empty} &= \lambda \mathsf{x}.\ \textbf{true} \\ \textbf{cons} &= \lambda \mathsf{h}\ \mathsf{t}.\ \textbf{pair}\ \mathsf{h}\ \mathsf{t} \end{aligned}$$

Definition: list operations:

$$\begin{aligned} \textbf{isEmpty} &= \lambda \mathsf{l}.\mathsf{l}\ (\lambda \mathsf{x}\ \lambda \mathsf{y}.\textbf{false}) \\ \textbf{head} &= \textbf{fst} \\ \textbf{tail} &= \textbf{snd} \end{aligned}$$

Local definitions

Programs are usually structured as

```
let a1 = exp1 in
let a2 = exp2 in
...
let aN = expN in
  main
```

By means of function application, we can easily represent (non-recursive) local definitions using the following scheme:

$$\textbf{let}\ \mathsf{a} = \mathsf{v}\ \textbf{in}\ \mathsf{p} \quad \mapsto \quad (\lambda \mathsf{a}.\mathsf{p})\ \mathsf{v}$$

Recursive functions

Let us assume we want to represent in lambda calculus the **factorial** function, defined below:

$$\mathsf{fact}(\mathsf{n}) = \begin{cases} 1 & \text{if } \mathsf{n} = 0 \\ \mathsf{n} * \mathsf{fact}(\mathsf{n} - 1) & \text{otherwise} \end{cases}$$

Notice: up to this point we know how to represent all the required auxiliary functions: test of zero, multiplication and predecessor.

Recursive functions (2)

Initial attempt:

$$\begin{array}{lll} \mathsf{let} & \mathsf{fact} = \lambda \mathsf{n}.\mathbf{if} & (\mathbf{isZero}\ \mathsf{n}) \\ & & \mathbf{1} \\ & & (\mathbf{mult}\ \mathsf{n}\ (\mathsf{fact}\ (\mathbf{pred}\ \mathsf{n}))) \\ \mathsf{in} & (\mathsf{fact}\ 3) & \end{array}$$

Problem: **it does not work!**

- there is no memory to remember that fact in the right-hand side is what we are defining (circular definition)
- in other words: fact in the right-hand side is a **free variable**!

Question: how can we obtain a lambda term that encodes the recursive definition of fact?

Recursive functions (3)

Imagine the RHS of the equation as the application of a *combinator* S over fact.

In this case, the equation

$$\mathsf{fact} = \lambda\, \mathsf{n}\, .\mathbf{if}\ (\mathbf{isZero}\ \mathsf{n})\ 1\ (\mathsf{mult}\ \mathsf{n}\ (\mathsf{fact}\ (\mathbf{pred}\ \mathsf{n})))$$

becomes

$$\mathsf{fact} = \mathsf{S}\ \mathsf{fact}$$

where

$$\mathsf{S} = \lambda\, \mathsf{R}.\lambda\, \mathsf{n}\, .\mathbf{if}\ (\mathbf{isZero}\ \mathsf{n})\ 1\ (\mathsf{mult}\ \mathsf{n}\ (\mathsf{R}\ (\mathbf{pred}\ \mathsf{n})))$$

In other words: we would like fact to be a **fixed point** of the S combinator.

Notice: the definition of S is **not** circular.

Fixed points

Definition: a **fixed point** (FP) of a function $f : X \to X$ is an element $x \in X$ such that $f(x) = x$

Example:

$$
\begin{array}{llll}
1) & x \mapsto x^2 & : \mathbb{N} \to \mathbb{N} & \text{FP} \in \{0, 1\} \\
2) & x \mapsto |x| & : \mathbb{Z} \to \mathbb{Z} & \text{FP} \in \{0, 1, 2, \ldots\} \\
3) & x \mapsto -x & : \mathbb{Z} \to \mathbb{Z} & \text{FP} \in \{0\} \\
4) & x \mapsto x - 1 & : \mathbb{Z} \to \mathbb{Z} & \text{FP} \in \{\}
\end{array}
$$

We want to find a fixed point for $S : \Lambda \to \Lambda$.

Good news! There are lambda terms that **build** a fixed point based on the combinator itself!

The $\mathbf{Y}$ combinator

The following term is the famous **Y combinator**:

$$\mathbf{Y} = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

Theorem: $\mathbf{Y}$ produces a fixed-point for its argument.

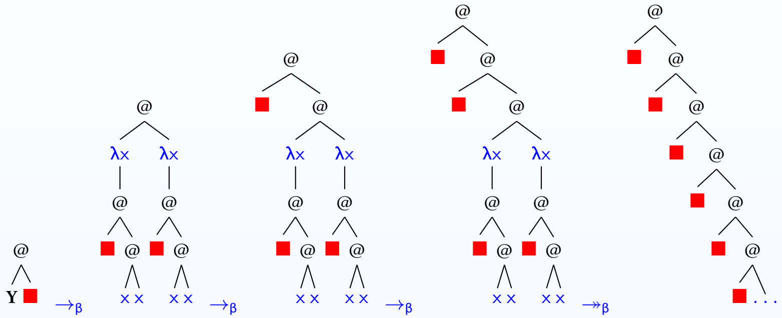
Proof:

$$\begin{aligned}
\mathbf{Y}\,S & \quad \to_\beta \\
(\lambda x.S(x\,x))\,(\lambda x.S(x\,x)) & \quad \to_\beta \\
S\,(\lambda x.S(x\,x))\,(\lambda x.S(x\,x)) & \quad =_\beta \\
S(\mathbf{Y}\,S) &
\end{aligned}$$

$\mathbf{Y}\,S =_\beta S(\mathbf{Y}\,S)$ and, consequently, $\mathbf{Y}\,S$ is a fixed point of S □

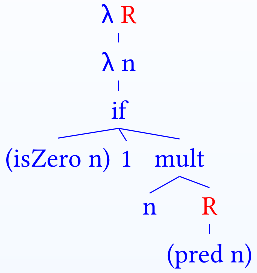
Using the $\mathbf{Y}$ combinator, we can finally define $\mathsf{fact} = \mathbf{Y}\,S$.

The Y combinator: example



The Y combinator: example (2)

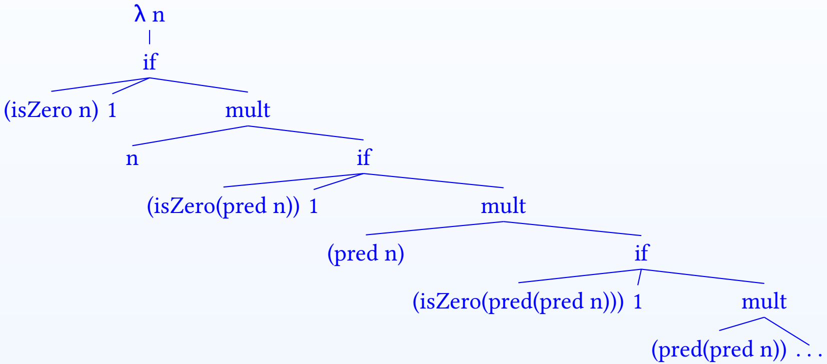
if $\mathsf{S} \;=_\beta$



Obs: for clarity, here we represent the arguments of the function **below** the node that represents the function call.

The Y combinator: example (3)

then $(\mathbf{Y}\ \mathsf{S}) =_\beta$



Length of a list

Notice that we only need to specify the **recursion pattern** and the fixed point combinator Y provides us with the lambda term associated with a **recursive definition**.

Example: recursive definition of the length of a list:

$$\mathsf{length}(\ell) = \begin{cases} 0 & \text{if } \ell = \textbf{empty} \\ 1 + \mathsf{length}(\mathsf{tail}(\ell)) & \text{otherwise} \end{cases}$$

Recursion pattern:

$$\begin{aligned} \mathsf{L} = \lambda \mathsf{R}.\lambda\ell.\textbf{if} \quad & (\textbf{isEmpty}\ \ell) \\ & \textbf{0} \\ & (\textbf{succ}\ (\mathsf{R}\ (\textbf{tail}\ \ell))) \end{aligned}$$

Lambda term: $\textbf{length} = \textbf{Y}\ \mathsf{L}$

Natural division

A version of integer division for natural numbers $\mathsf{div}(\mathsf{a}, \mathsf{b})$ (returning 0 if b is 0) can be defined recursively as follows:

$$\mathsf{div}(\mathsf{a}, \mathsf{b}) = \begin{cases} 0 & \text{if } \mathsf{a} < \mathsf{b} \text{ or } \mathsf{b} = 0 \\ 1 + \mathsf{div}(\mathsf{a} - \mathsf{b}, \mathsf{b}) & \text{otherwise} \end{cases}$$

Recursion pattern:

$$\begin{aligned} \mathsf{D} = \lambda \mathsf{R}.\lambda \mathsf{a}.\lambda \mathsf{b}.\mathbf{if}\ & (\mathbf{lt}\ \mathsf{a}\ \mathsf{b}) \\ & \mathbf{0} \\ & (\mathbf{succ}\ (\mathsf{R}\ (\mathbf{sub}\ \mathsf{a}\ \mathsf{b})\ \mathsf{b})) \end{aligned}$$

Lambda term: $\mathbf{div} = \lambda \mathsf{a}.\lambda \mathsf{b}.\mathbf{if}\ (\mathbf{isZero}\ \mathsf{b})\ \mathbf{0}\ (\mathbf{Y}\ \mathsf{D}\ \mathsf{a}\ \mathsf{b})$

Algebraic datatypes

An **algebraic data type** is a type whose elements are built from a finite combination of calls to *parameterized constructors*.

Many datatypes seen previously are algebraic (booleans, naturals, etc.)

Example: (in Haskell syntax)

```
data Bool = True | False
```

Some parameters of constructors may be of the same type as the type being defined.

Example:

```
data Nat = Zero | Succ Nat
data ListInt = Empty | Cons Int ListInt
```

Scott encoding of algebraic datatypes

There is a standard way of encoding any algebraic datatype in the pure lambda calculus, which is known as **Scott encoding**.

Assume a datatype with n constructors, each represented by C_i for $1 \leq i \leq n$. Assume constructor C_i has m_i arguments, named $a_1 \ldots a_{m_i}$.

Each *fully parameterized* constructor C_i is represented by a term of the format

$$\lambda c_1\ c_2\ \ldots\ c_n\ .\ c_i\ a_1\ a_2\ \ldots\ a_{m_i}$$

Each *constructor* is represented by a function that receives the constructor arguments a_i, and returns a fully parameterized constructor in the format above.

$$\lambda a_1 \cdots a_{m_i}\ .\ \lambda c_1\ c_2\ \ldots\ c_n\ .\ c_i\ a_1\ a_2\ \ldots\ a_{m_i}$$

Scott encoding: examples

```
data Bool = True | False
```

$$\begin{aligned} \mathsf{true} &= \lambda t\ f.t \\ \mathsf{false} &= \lambda t\ f.f \end{aligned}$$

```
data Nat  = Succ Nat | Zero
```

$$\begin{aligned} \mathsf{zero} &= \lambda s\ z.z \\ \mathsf{succ} &= \lambda n.\lambda s\ z.s\ n \end{aligned}$$

```
data ListInt = Empty
             | Cons Int ListInt
```

$$\begin{aligned} \mathsf{empty} &= \lambda e\ c.e \\ \mathsf{cons} &= \lambda h\ t.\lambda e\ c.c\ h\ t \end{aligned}$$

Notice that the Scott encoding is the same as the Church encoding of booleans. For natural number and lists, however, they are quite distinct.

Scott encoding: tests

Tests can be implemented by passing as arguments for the *fully parameterized constructors* functions that receive the arguments of each constructor, and return **true** or **false** accordingly.

$$\begin{aligned} \mathsf{isTrue} &= \lambda b.b\ \mathbf{true}\ \mathbf{false} \\ \mathsf{isFalse} &= \lambda b.b\ \mathbf{false}\ \mathbf{true} \end{aligned}$$

$$\begin{aligned} \mathsf{isZero} &= \lambda n.n\ (\lambda x.\mathbf{false})\ \mathbf{true} \\ \mathsf{isSucc} &= \lambda n.n\ (\lambda x.\mathbf{true})\ \mathbf{false} \end{aligned}$$

$$\begin{aligned} \mathsf{isEmpty} &= \lambda l.l\ \mathbf{true}\ (\lambda h\ t.\mathbf{false}) \\ \mathsf{isCons} &= \lambda l.l\ \mathbf{false}\ (\lambda h\ t.\mathbf{true}) \end{aligned}$$

Scott encoding: destructors

Destructors can be implemented by passing selector functions as arguments for the *fully parameterized constructor*.

Each destructor focus on a specific constructor: for the other arguments, an error term (such as **false**) or some arbitrary default value can be used.

$$
\begin{array}{rcl}
\mathsf{predSucc} & = & \lambda \mathsf{n}.\mathsf{n}\ (\lambda \mathsf{x}.\mathsf{x})\ \mathbf{false} \\[1em]
\mathsf{headCons} & = & \lambda \mathsf{l}.\mathsf{l}\ \mathbf{false}\ (\lambda \mathsf{h}\ \mathsf{t}.\mathsf{h}) \\
\mathsf{tailCons} & = & \lambda \mathsf{l}.\mathsf{l}\ \mathbf{false}\ (\lambda \mathsf{h}\ \mathsf{t}.\mathsf{t})
\end{array}
$$

Notice: only constructors with arguments require associated destructors.

Exercise: define the algebraic datatype for binary trees of integers, and develop its Scott encoding.

Properties of the lambda calculus

We will now state and prove (or at least sketch the structure of the proof) of the two important properties of the β-reduction relation that we have been assuming so far:

- Confluence (Church-Rosser)
- Normalization

As mentioned before, these properties are important when considering lambda calculus as a model of computation.

We also want to prove that lambda-calculus is a universal model of computation, at least as powerful as Turing-machines:

- Turing-completeness

Confluence (Church-Rosser property)

Theorem: (Confluence) if $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then there is a term P such that $N_1 \twoheadrightarrow_\beta P$ and $N_2 \twoheadrightarrow_\beta P$.

Corolary: normal forms are unique.

Example:

$$\begin{tikzcd}
\boldsymbol{\omega}(\mathbf{II}) \arrow[rr] \arrow[d] & & \boldsymbol{\omega}(\mathbf{I}) \arrow[d] \\
(\mathbf{II})(\mathbf{II}) \arrow[r] & \mathbf{I}(\mathbf{II}) \arrow[r] & \mathbf{II}
\end{tikzcd}$$

The proof of this theorem requires a new relation: *parallel reduction.*

Parallel reduction

Definition: (parallel reduction) $\Rightarrow_\beta$ is the smallest relation on Λ such that

$$x \Rightarrow_\beta x \quad (\textsc{parVar}) \qquad \frac{M \Rightarrow_\beta M' \qquad N \Rightarrow_\beta N'}{M\ N \Rightarrow_\beta M'\ N'} \ (\textsc{parApp})$$

$$\frac{M \Rightarrow_\beta M'}{\lambda x.M \Rightarrow_\beta \lambda x.M'} \ (\textsc{parAbs}) \qquad \frac{P \Rightarrow_\beta P' \qquad Q \Rightarrow_\beta Q'}{(\lambda x.P)\ Q \Rightarrow_\beta [Q'/x]P'} \ (\textsc{parBeta})$$

Lemma:

(a) if $M \to_\beta N$ then $M \Rightarrow_\beta N$

(b) if $M \Rightarrow_\beta N$ then $M \twoheadrightarrow_\beta N$

(c) if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$ then $[N/x]M \Rightarrow_\beta [N'/x]M'$

Proof: structural induction in $M \to_\beta N$ (a), $M \Rightarrow_\beta N$ (b) and $M \Rightarrow_\beta M'$ (c).

Complete reduction

Definition: (complete reduction) $\_^* : \Lambda \to \Lambda$ is defined as

$$\begin{array}{ll} x^* & = x \\ (\lambda x.M)^* & = \lambda x.(M^*) \\ (M\ N)^* & = \begin{cases} M^*\ N^* & \text{if } M \neq (\lambda y.P) \\ [N^*/y]P^* & \text{if } M = (\lambda y.P) \end{cases} \end{array}$$

Notice:

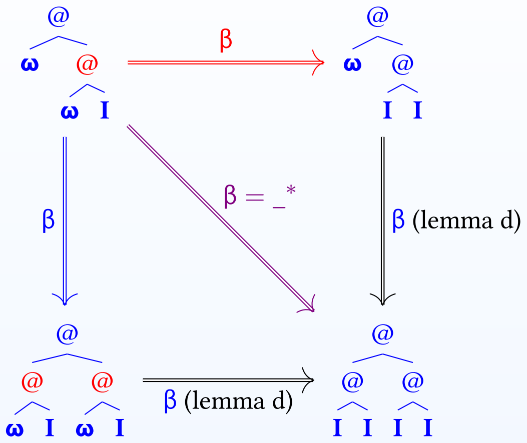
- $M \Rightarrow_\beta N$ if N is obtained by reducing **some** redexes in M;
- M^* is obtained reducing **all** redexes in M.

Lemma: (d) if $M \Rightarrow_\beta N$ then $N \Rightarrow_\beta M^*$.

Proof: structural induction in $M \Rightarrow_\beta N$ using lemmas (a), (b) and (c)

Confluence (example)

Example:



Confluence (proof)

Theorem: If $\mathsf{M} \twoheadrightarrow_\beta \mathsf{N}$ and $\mathsf{M} \twoheadrightarrow_\beta \mathsf{N}'$ then exists P such that $\mathsf{N} \twoheadrightarrow_\beta \mathsf{P}$ and $\mathsf{N}' \twoheadrightarrow_\beta \mathsf{P}$.

Proof:

$$\begin{tikzcd}
\mathsf{M} \arrow[r, two heads] \arrow[d, two heads] & \bullet \arrow[r, two heads] \arrow[d, two heads] & \mathsf{N} \arrow[d, two heads] \\
\bullet \arrow[r, two heads] \arrow[d, two heads] & \bullet \arrow[r, two heads] \arrow[d, two heads] & \bullet \arrow[d, two heads] \\
\mathsf{N}' \arrow[r, two heads] & \bullet \arrow[r, two heads] & \bullet
\end{tikzcd}$$

1. If $\mathsf{M} = \mathsf{N}$ or $\mathsf{M} = \mathsf{N}'$, trivial
2. Otherwise, there are finite sequences $\mathsf{M} \to_\beta \ldots \to_\beta \mathsf{N}$ and $\mathsf{M} \to_\beta \ldots \to_\beta \mathsf{N}'$.
3. Lemma (a) $(\to_\beta \text{ to } \Rightarrow_\beta)$.
4. Lemma (d) (confluence of $\Rightarrow_\beta$).
5. Lemma (b) $(\Rightarrow_\beta \text{ to } \twoheadrightarrow_\beta)$.
6. Transitivity of $\twoheadrightarrow_\beta$. $\square$

Corolary: If $\mathsf{N} \leftarrow_\beta \mathsf{M} \to_\beta \mathsf{N}'$ then $\mathsf{N} \twoheadrightarrow_\beta \mathsf{P} \twoheadleftarrow_\beta \mathsf{N}'$.

Existence of normalizing strategies

We say that an *evaluation strategy* $\rightarrow_{\mathsf{e}} \subseteq \Lambda \times \Lambda$ is **normalizing** when the following holds: if $M =_{\beta} N$, and N is a normal form, then $M \twoheadrightarrow_{\mathsf{e}} N$.

Counterexample: the applicative strategy $\rightarrow_{\mathsf{a}}$ is **not** normalizing (as seen previously).

Notice: some terms do not have normal forms: even normalizing strategies do not arrive at normal forms when they do not exist.

Theorem: the *normal strategy* $\rightarrow_{\mathsf{n}}$ (i.e. always choose the leftmost, outermost redex) is normalizing.

To prove this property, we must introduce a new concept: *head normal forms.*

Head normal form

Notation: Let us use $\lambda\vec{x}.M$ as an abbreviation for $\lambda x_1\ x_2 \ldots\ x_n.M$, and define $|\vec{x}| = n$ (the number of lambdas).

Let us also use $M\ \vec{N}$ as an abbreviation for a sequence of applications $M\ N_1\ N_2\ \ldots\ N_n$, and, similarly, define $|\vec{N}| = n$ (the number of applications/arguments).

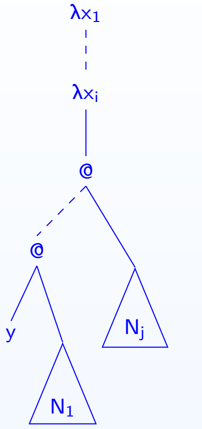
Lemma: each lambda term M is in one of the two following forms (exclusively):

- **head normal form:** $\lambda\vec{x}.\ y\ \vec{N}$, with $|\vec{x}| \geq 0$ and $|\vec{N}| \geq 0$.
  Notation: $HNF(M)$. The variable y is called the **head variable**.

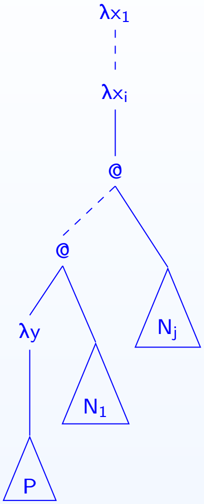
- **head-reducible term:** $\lambda\vec{x}.\ (\lambda y.P)\ \vec{N}$, with $|\vec{x}| \geq 0$ and $|\vec{N}| > 0$.
  Notation: $HR(M)$. The redex $(\lambda y.P)\ N_1$ is named **head redex**, and all other redexes are called **(head) internal**.

Proof: by induction in M.

Head normal form vs head-reducible term



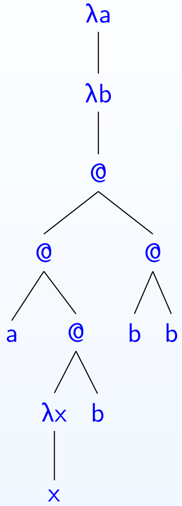
head normal form, $i, j \geq 0$



head-reducible term, $i \geq 0, j > 0$

Head normal form vs normal form

Example:



The *head redex* (when it exists) is always the leftmost, outermost redex.

However, the leftmost, outermost redex is not always a head redex, as the example shows.

Therefore:

- every *normal form* is in *head normal form*;
- some *head normal forms* are not *normal forms*.

Head reduction

Let M be a head-reducible term. Denote $M \to_h N$ the β-reduction of the head redex (head reduction).

We write $M \xrightarrow{*}_h N$ or $M \twoheadrightarrow_h N$ to denote its reflexive and transitive closure.

Notice: a sequence of head reductions can either

- diverge, or
- stop at a *head normal form*

Parallel internal reduction

The relation $\mathsf{M} \Rightarrow_{\mathsf{i}} \mathsf{N}$ denotes the parallel reduction of 0 or more (head) internal redexes of M. Let us denote $\mathsf{M} \overset{*}{\Rightarrow}_{\mathsf{i}} \mathsf{N}$ its reflexive and transitive closure.

Definition: $\Rightarrow_{\mathsf{i}}$ is the smallest relation satisfying

$$\frac{\mathsf{P}_1 \Rightarrow_\beta \mathsf{P}_1' \quad \cdots \quad \mathsf{P}_n \Rightarrow_\beta \mathsf{P}_n' \qquad m \geq 0 \qquad n \geq 0}{\begin{array}{c}(\lambda x_1 \cdots x_m.y\ \mathsf{P}_1 \cdots \mathsf{P}_n) \\ \Rightarrow_{\mathsf{i}} \\ (\lambda x_1 \cdots x_m.y\ \mathsf{P}_1' \cdots \mathsf{P}_n')\end{array}}$$

$$\frac{\mathsf{P}_0 \Rightarrow_\beta \mathsf{P}_0' \qquad \mathsf{P}_1 \Rightarrow_\beta \mathsf{P}_1' \quad \cdots \quad \mathsf{P}_n \Rightarrow_\beta \mathsf{P}_n' \qquad m \geq 0 \qquad n \geq 1}{\begin{array}{c}(\lambda x_1 \cdots x_m.(\lambda y.\mathsf{P}_0)\ \mathsf{P}_1 \cdots \mathsf{P}_n) \\ \Rightarrow_{\mathsf{i}} \\ (\lambda x_1 \cdots x_m.(\lambda y.\mathsf{P}_0')\ \mathsf{P}_1' \cdots \mathsf{P}_n')\end{array}}$$

Requirements for the proof of normalization

The proof of normalization requires some lemmas involving head reductions and parallel internal reductions (assumed as true for now, we will prove them later).

Lemma: (A) if $M \Rightarrow_i M'$ and $M' = \lambda\vec{x}.\ y\ \vec{N'}$ then $M = \lambda\vec{x}.\ y\ \vec{N}$ (such that $N_i \Rightarrow_\beta N_i'$).

Intuition: $\Rightarrow_i$ preserves the type of both terms (HNF or HR).

Lemma: (B) if $M \Rightarrow_\beta N$, then exists P such that $M \twoheadrightarrow_h P \Rightarrow_i N$.

Intuition: a beta parallel reduction can be decomposed into a sequence of head reductions, followed by a single parallel internal reduction.

Lemma: (C) if $M \Rightarrow_i N \rightarrow_h P$ then exists N' such that $M \twoheadrightarrow_h N' \Rightarrow_i P$.

Intuition: a single head reduction after a parallel internal reduction can be *pushed to the left* as a *sequence* of head reductions.

Requirements for the proof of normalization (2)

Lemma: (D) If $M \twoheadrightarrow_{\beta} N$, then exists P such that $M \twoheadrightarrow_{h} P \overset{*}{\Rightarrow}_{i} N$.

Proof:

1. Sequence of β-reductions
2. $\rightarrow_{\beta}$ to $\Rightarrow_{\beta}$
3. Lemma (B)
4. Lemma (C) $\square$

$$\begin{tikzcd}
M \arrow[rr, Rightarrow, "\beta"] \arrow[dr, twoheadrightarrow, "h"'] & & \bullet \arrow[rr, Rightarrow, "\beta"] \arrow[dr, twoheadrightarrow, "h"'] & & \bullet \arrow[rr, Rightarrow, "\beta"] \arrow[dr, twoheadrightarrow, "h"'] & & N \\
& \bullet \arrow[ur, Rightarrow, "i"'] \arrow[dr, twoheadrightarrow, "h"'] & & \bullet \arrow[ur, Rightarrow, "i"'] \arrow[dr, twoheadrightarrow, "h"'] & & \bullet \arrow[ur, Rightarrow, "i"'] & \\
& & \bullet \arrow[ur, Rightarrow, "i"'] \arrow[dr, twoheadrightarrow, "h"'] & & \bullet \arrow[ur, Rightarrow, "i"'] & & \\
& & & P \arrow[ur, Rightarrow, "i"'] & & &
\end{tikzcd}$$

The normal strategy is normalizing

Theorem: If N is normal form and $M \twoheadrightarrow_\beta N$, then $M \twoheadrightarrow_n N$.

Proof: by induction on the **size** of normal form N.

1. We know that $N = (\lambda x_1 \cdots x_m.\, y\, N_1 \;\cdots\; N_n)$ is a head normal form;
2. There is P such that $M \twoheadrightarrow_h P \overset{*}{\Rightarrow}_i N$ (Lemma D);
3. $P = (\lambda x_1 \cdots x_m.\, y\, P_1 \;\cdots\; P_n)$ and $P_i \overset{*}{\Rightarrow}_\beta N_i$ for each $1 \leq i \leq n$ (Lemma A);
4. Therefore, we have $P_i \twoheadrightarrow_\beta N_i$ for $1 \leq i \leq n$, where each N_i is a normal form *strictly smaller* than N. By the inductive hypothesis, we have $P_i \twoheadrightarrow_n N_i$.
5. Finally, the reduction using only leftmost, outermost redexes is:

$$\begin{array}{rl} M \twoheadrightarrow_h & (\lambda x_1 \cdots x_m.\, y\, P_1\, P_2 \;\cdots\; P_n) = P \\ \twoheadrightarrow_n & (\lambda x_1 \cdots x_m.\, y\, N_1\, P_2 \;\cdots\; P_n) \\ \twoheadrightarrow_n & (\lambda x_1 \cdots x_m.\, y\, N_1\, N_2 \;\cdots\; P_n) \\ \twoheadrightarrow_n & \cdots \\ \twoheadrightarrow_n & (\lambda x_1 \cdots x_m.\, y\, N_1\, N_2 \;\cdots\; N_n) = N \end{array}$$

Universality

Theorem: lambda calculus is a Turing-complete model.

Proof: (sketch)

- Using two lists we can represent a bidirectional tape with a moving read/write head;
- States and symbols can be represented by numbers;
- Using lists of tuples we can represent the state transition function and an associated recursive lookup function;
- A tuple containing the current state, a tape and the state transition function represents a configuration;
- A recursive function `run` over configurations may be defined: if the received configuration is final, it is returned; otherwise a recursive call over the next configuration (calculated from the received one) is performed.

Therefore, it is possible to simulate any Turing machine in lambda calculus. □

Undecidability of beta-reduction

Theorem: determining if $\mathsf{A} =_\beta \mathsf{B}$ is **undecidable**.

Proof: by a reduction from the halting problem for Turing Machines. **Halting problem:** given (M, w) a pair such that M is a TM and w an input word for M, is it the case that the computation of M over w halts?

- let isFinalCfg be the lambda term that tests if a configuration (of a Turing Machine) is final (either accepting or rejecting) or not, returning a boolean value accordingly;
- let run be the lambda term that simulates a Turing machine configuration until it reaches a final configuration, diverging otherwise;
- let $[\mathsf{M}, \mathsf{w}]$ be the encoding of the initial configuration of (M, w) as a lambda term;
- let $\mathsf{A} = \mathsf{isFinalCfg}\ (\mathsf{run}\ [\mathsf{M}, \mathsf{w}])$ and $\mathsf{B} = \textbf{true}$

If (M, w) halts, then $\mathsf{A} =_\beta \textbf{true}$ and $\mathsf{A} =_\beta \mathsf{B}$. Otherwise, $\mathsf{run}\ [\mathsf{M}, \mathsf{w}]$ nevers reaches a final configuration (or normal form), and $\mathsf{isFinalCfg}\ (\mathsf{run}\ [\mathsf{M}, \mathsf{w}]) \neq_\beta \textbf{true}$. Therefore, $\mathsf{A} \neq_\beta \mathsf{B}$.

In conclusion, a method for determining β-equivalence would give us a method for solving the halting problem for TMs (which is known to be undecidable). □

Lambda calculus: review

- Lambda calculus models the definition and application of functions;
- Lambda terms are equivalence classes of α-equivalent pre-terms;
- The β-reduction of terms is built around the notion of substitution;
- The β-reduction relation is confluent;
- The normal evaluation strategy is normalizing;
- Many data types and control structures can be encoded in the pure lambda calculus;
- We can create lambda terms that represent functions based on their recursive definition using fixed point operators;
- The lambda calculus is a Turing-complete model of computation;
- Determining if two terms A and B are β-equivalent is undecidable.

Content

Simply Typed Lambda Calculus

Adding types

We saw that the abstract behaviour of functions can be expressed very well by means of λ-calculus

However, how functions of pure λ-calculus act on their input λ-calculus is ‘too liberal’

Introducing types is a natural thing to do!

Adding **simple types** to λ-calculus we obtain Simple Typed Lambda Calculus (STLC or $\lambda\to$ for short)

After STLC (or $\lambda\to$) our next calculus, $\lambda 2$, also have polymorphic types

Adding types

The syntax of $\lambda{\to}$ is given in stages:

1. We define the types
2. We define the set of pre-typed terms, which are
   - exactly the terms of λ, if we choose $\lambda{\to}$ in Curry style, or
   - the terms of λ with some type annotation, if we choose $\lambda{\to}$ in Church style
3. We define a type system that identifies all typed terms of $\lambda{\to}$.

As expected, terms with self-application, among others, will be excluded from $\lambda{\to}$ the type system

Syntax of $\lambda\!\rightarrow$ types

We start with

- an arbitrary non-empty set $\mathbb{A}$ of atomic types and
- a type constructor $\rightarrow$ for function types

We will use letters $\mathsf{A}, \mathsf{B}, \ldots \mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$, and variants to represent types in $\mathbb{A}$.

Definition: The set of simple types $\mathbb{T}$ is the smallest set such that:

1. If $\mathsf{X} \in \mathbb{A}$ then $\mathsf{X} \in \mathbb{T}$
2. If $\tau_1, \tau_2 \in \mathbb{T}$ then $\tau_1 \rightarrow \tau_2 \in \mathbb{T}$

The same set $\mathbb{T}$ can be defined using an abstract grammar:

Definition: The set $\mathbb{T}$ of simple types is defined by the following abstract grammar:

$$\tau \ ::= \ \mathsf{X} \ \mid \ \tau_1 \rightarrow \tau_2$$

Syntax of $\lambda{\rightarrow}$ types

For our purposes, all that matters about the types in $\mathbb{A}$ is that they are **atomic**. Examples of concrete atomic types are `nat`, `bool`, `char`

We could add concrete atomic types and other type constructors such as one for product types, for instance, but for now let us consider the simplest version of this calculus.

The type constructor $\rightarrow$ is right associative, hence a type like

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$$

is to be read as $\tau_1 \rightarrow (\tau_2 \rightarrow (\tau_3 \rightarrow \tau_4))$

Examples of types:

X	$(X \rightarrow X) \rightarrow Y$	$((X \rightarrow X) \rightarrow X) \rightarrow X$
$X \rightarrow Y$	$X \rightarrow (X \rightarrow Y)$	$(X \rightarrow X) \rightarrow (X \rightarrow X)$
$X \rightarrow X$	$X \rightarrow X \rightarrow Y$	$(X \rightarrow X) \rightarrow X \rightarrow X$

Syntax of $\lambda\!\to$ Church-Style

In the Church-style, variables are declared with their types in λ abstractions.

Definition: The set $\Lambda_{\mathbb{T}}^{-}$ of pre-typed preterms is defined by the following abstract grammar:

$$M ::= x \mid M\,N \mid \lambda x{:}\tau.\,M$$

Example:

(a)	$\lambda x{:}A.\,x$	of type $A \to A$
(b)	$\lambda x{:}A.\,\lambda z{:}A \to B.\,(z\,x)$	of type $A \to ((A \to B) \to B)$
(c)	$\lambda x{:}A \to A.\,(x\,x)$	type error
(d)	$\lambda x{:}A.\,(y\,x)$	depends on a typing context

The set $\Lambda_{\mathbb{T}} = \Lambda_{\mathbb{T}}^{-}/ =_{\alpha}$ of pre-typed terms correspond to the set of pre-typed preterms modulo alpha-equivalence. The definition of $=_{\alpha}$ is very similar to the one the untyped lambda calculus and therefore it is omitted here.

Type system for $\lambda{\to}$

A typing context is a function mapping a finite set of variables to types.

Typing contexts can be written as $x_1 : A_1, x_2 : A_2, \ldots x_n : A_n$

We use metavariables $\Gamma, \Gamma', \ldots$ to represent typing contexts

We now define a ternary *typing relation* between contexts, pre-typed terms and types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad (\textsc{var})$$

$$\frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M\, N : \tau'} \qquad (\textsc{app})$$

$$\frac{\Gamma,\ x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x : \tau.\, M : \tau \to \tau'} \qquad (\textsc{abs})$$

Context $\Gamma, x : \tau$ is the same as Γ except that it maps x to τ.

Type derivation

Here is an example of a *typing derivation* (Where $\Gamma = y : A \to B, z : A$)

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(y) = A \to B}{\Gamma \vdash y : A \to B}\,\text{VAR} \qquad \dfrac{\Gamma(z) = A}{\Gamma \vdash z : A}\,\text{VAR}}{\Gamma \vdash y\ z : B}\,\text{APP}}{y : A \to B \vdash \lambda z : A.\ y\ z : A \to B}\,\text{ABS}}{\vdash \lambda y : A \to B.\ \lambda z : A.\ y\ z : (A \to B) \to A \to B}\,\text{ABS}$$

Observe that there is exactly one typing rule for each syntactic structure (variable, application, and abstraction)

Hence, typing derivations are determined by the syntactic structure of terms

Typing problems

The following are algorithmic typing problems of interest:

1. Well-typedness - given M, is there Γ and τ s.t. $\Gamma \vdash \mathsf{M} : \tau$?
2. Type assignment - given M and Γ, is there τ s.t. $\Gamma \vdash \mathsf{M} : \tau$?
3. Type checking - given M, Γ, and τ, does $\Gamma \vdash \mathsf{M} : \tau$ hold?
4. Inhabitation - given Γ and τ, is there term M s.t. $\Gamma \vdash \mathsf{M} : \tau$?

All these problems are decidable in $\lambda{\rightarrow}$, i.e. there are algorithms that answer these questions and produce context, terms and types.

Definition: a term $\mathsf{M} \in \Lambda_{\mathbb{T}}$ is called **legal** or **typable** if there are Γ and τ such that $\Gamma \vdash \mathsf{M} : \tau$.

Properties of the type system

We can simplify proofs if we adopt the Barendregt assumption. By that assumption, all variables in typing contexts are distinct. In some proofs, it might also be useful to treat typing context as lists.

Lemma: (Free Variables Lemma) If $\Gamma \vdash M : \tau$ then $FV(M) \subseteq \mathtt{dom}(\Gamma)$

Lemma: (Weakening, Condensing, Permutation)

1. (Weakening) If $\Gamma' \vdash M : \tau$ and $\Gamma' \subseteq \Gamma''$, then $\Gamma'' \vdash M : \tau$
2. (Condensing) If $\Gamma \vdash M : \tau$ then $\Gamma \downarrow_{FV(M)} \vdash M : \tau$
3. (Permutation) If $\Gamma \vdash M : \tau$ and Γ' is a permutation of Γ then $\Gamma' \vdash M : \tau$

Lemma: (Uniqueness of Types) If $\Gamma \vdash M : \tau$ and $\Gamma \vdash M : \tau'$ then $\tau = \tau'$

Lemma: (Substitution) If $\Gamma, x : \tau \vdash M : \tau'$ and $\Gamma \vdash N : \tau$, then $\Gamma \vdash [N/x]\, M : \tau'$

Theorem: (Decidability of Typing Problems) Well-typedness, type assignment, type-checking, and type inhabitation for $\lambda\!\rightarrow$ are all decidable problems

Beta reduction: formal definition

The rules are **the same** as those of untyped lambda calculus:

Definition: $\rightarrow_{\boldsymbol{\beta}}$ is the least relation on $\Lambda_{\mathbb{T}}$ such that

$$(\lambda x : \tau .\ P)\ Q \rightarrow_\beta [Q/x]P \qquad (\beta) \qquad\qquad \frac{M \rightarrow_\beta M'}{N\ M \rightarrow_\beta N\ M'}$$

$$\frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'} \qquad\qquad \frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N}$$

Definition: $\twoheadrightarrow_{\boldsymbol{\beta}}$ (or $\overset{*}{\rightarrow}_\beta$) is the reflexive and transitive closure of $\rightarrow_\beta$.

Properties of beta reduction

Theorem: (Confluency) The Church-Rosser property holds for $\lambda\!\rightarrow$.

Theorem: (Subject Reduction) If $\Gamma \vdash M : \tau$ and $M \twoheadrightarrow_{\beta} M'$ then $\Gamma \vdash M' : \tau$.
Intuitively: the type of a legal term is preserved by reductions.

Theorem: (Weak Normalisation) If there are Γ and τ such that $\Gamma \vdash M : \tau$ then M is weakly normalising.
Intuitively: there is a reduction path starting in M that reaches a normal form.

Theorem: (Strong Normalisation) If there are Γ and τ such that $\Gamma \vdash M : \tau$ then M is strongly normalising.
Intuitively: all reduction paths starting in M are finite and reach a normal form.

$\lambda\!\rightarrow$ is less expressive than λ (it does not admit self-application and the Y combinator cannot be typed)

Normalizing terms

A term M is *weakly normalizing* (written WN(M)) if it has a normal form (i.e. some reduction paths from M terminate).

A term M is *strongly normalizing* (written SN(M)) if it does not have any infinite reduction path (i.e. all reduction paths from M terminate).

To clarify the distinction, consider the following terms in the *untyped lambda calculus*:

- (λx.x x) (λx.x x)
- (λx.x x x) (λx.x)
- (λx.x x x) (λx.x x x)
- (λx.y) ((λx.x x x) (λx.x x x))

Exercise: for each of the terms above, determine if it is weakly normalizing, strongly normalizing or neither.

Formula of weak normalization

The ideia of $\mathsf{WN}(\mathsf{M})$ as "some execution paths starting in M terminate" can be represented by the following first-order formula:

$$\mathsf{WN}(\mathsf{M}) \quad \overset{\text{def}}{=} \quad \exists \mathsf{N}, (\ \mathsf{M} \twoheadrightarrow_\beta \mathsf{N}\ \land\ \neg\exists \mathsf{P}, \mathsf{N} \rightarrow_\beta \mathsf{P})$$

Example:

$$\begin{tikzcd}
& \bullet \arrow[r] \arrow[dr] & \bullet \arrow[r] & \mathsf{N} \\
\mathsf{M} \arrow[ur] \arrow[dr] & & \bullet \arrow[ur] \arrow[r] & \cdots \\
& \bullet \arrow[r] & \bullet \arrow[r] & \cdots
\end{tikzcd}$$

Normalization of all legal terms

We say that a typed calculus has the weak (resp. strong) normalization property if all legal (typable) terms are weakly (resp. strongly) normalizing.

Definition: $\lambda\!\rightarrow$ has the **weak normalization** property iff for all $M \in \Lambda_{\mathbb{T}}$,

$$\Gamma \vdash M : \tau \quad \Rightarrow \quad \mathsf{WN}(M)$$

Definition: $\lambda\!\rightarrow$ has the **strong normalization** property iff for all $M \in \Lambda_{\mathbb{T}}$,

$$\Gamma \vdash M : \tau \quad \Rightarrow \quad \mathsf{SN}(M)$$

Strategy for proving weak normalization

We now will prove that $\lambda\!\rightarrow$ (Church-style) has weak normalization.

This proof assigns a **measure** $\mu(M)$ to each well-typed term M. Then, we show that it is always possible to find (if M is not normal form) a reduction

$$M \rightarrow_\beta M'$$

such that $\mu(M') < \mu(M)$. This considers an ordering $(<)$ that is well-founded, i.e. in which there is no infinite decreasing chain of values.

Starting with M, by iterative reduction we obtain a strictly decreasing chain of measures

$$\mu(M) > \mu(M') > \mu(M'') \cdots$$

which entails that this particular sequence of reductions is finite.

This measure will be based on *degrees* of *types* and *redexes*, to be defined next.

Degrees

Definition: the **degree of a type** τ, denoted $\partial(\tau)$, is defined as

- $\partial(X) = 1$
- $\partial(\tau_1 \to \tau_2) = \max(\partial(\tau_1), \partial(\tau_2)) + 1$

Definition: the **degree of a redex** $r = (\lambda x{:}\tau_1.P)\ Q$, denoted $\partial(r)$, is

$$\partial(\tau_1 \to \tau_2)$$

where $\tau_1 \to \tau_2$ is the type of abstraction $(\lambda x{:}\tau_1.P)$.

Definition: the **degree of a term** M, denoted $d(M)$, is

- 0 if M is a normal form
- maximum value of $\partial(r)$ considering all possible redexes $r \in M$

Notice: a term $M = (\lambda x{:}\tau_1.P)\ Q$ has two degrees: one as a redex $\partial(M)$ and one as a term $d(M)$, in which the following holds: $\partial(M) \leq d(M)$.

Degrees, substitution and reduction

Lemma: If $x : \tau$ then $d([N/x]M) \leq \max(d(M), d(N), \partial(\tau))$. Proof: by case analysis of the effect of reduction on redexes.

Lemma: If $M \to_\beta M'$ then $d(M) \geq d(M')$. Proof: by induction on $M \to_\beta M'$.

The lemma above says that, if we have an arbitrary chain of reductions

$$M \to_\beta M' \to_\beta M'' \to_\beta \cdots \to_\beta N$$

we ensure $d(M) \geq d(M') \geq d(M'') \geq \cdots \geq d(N)$

This does not suffice, however, since we need $>$ instead of $\geq$:

$$\mu(M) > \mu(M') > \mu(M'') > \cdots > \mu(N)$$

We will show that is possible to **choose** a sequence of reductions with strictly decreasing measure.

Measure of a term

Definition: the **measure** of M, written $\mu(M)$, is the pair $(p, q) \in \mathbb{N} \times \mathbb{N}$, where

- $p = d(M)$ is the maximum degree of a redex in M
- q is the number of redexes of degree p in M

Special case: the measure of a normal form is $(0, 0)$.

Definition: we say $(a_1, b_1) < (a_2, b_2)$ when

$$(a_1 < a_2) \quad \vee \quad (a_1 = a_2 \wedge b_1 < b_2)$$

Notice: every strictly descending chain starting in an arbitrary pair (p, q) has **finite length**

Notice: unlike natural numbers, there is no bound to the length of a decreasing chain of pairs (although all chains are finite). There will be a bound, however, when we consider the length of reduction sequences from well-typed terms. This will be used later for proving strong normalization.

Reduction of maximum degree

Definition: **reduction of maximum degree:** given a term M, choose a redex $r_0 = (\lambda x : \tau_1 . P)\ Q$ such that

- $\partial(r_0) = d(M)$ (maximum redex degree)
- $d(P) < \partial(r_0)$ and $d(Q) < \partial(r_0)$ (inner redexes have smaller degree)

Lemma: if $M \to_\beta M'$ by reducing r_0 (with the properties above), then $\mu(M) > \mu(M')$.

Proof: for such particular r_0 we can guarantee that the degree of r_0 is *strictly greater* than the one of its contractum. Assume $\mu(M) = (a, b)$.

- If $b = 1$, then r_0 is the last redex of degree a in M, and after reduction $\mu(M') = (a', b')$ (for some a' and b'), with $a' < a$.
- If $b > 1$, there are other redexes of degree a in M, then reducing r_0 entails $\mu(M') = (a, b - 1)$.

In both cases, $\mu(M) > \mu(M')$. □

Weak normalization of $\lambda{\to}$

Theorem: **(weak normalization of $\lambda{\to}$)** if $\Gamma \vdash M : \tau$ then $\mathsf{WN}(M)$.

Proof: start a reduction path from M, always choosing a reduction of maximum degree (as defined in the previous slide).

$$M \xrightarrow{r_0} M' \xrightarrow{r_0'} \cdots$$

The corresponding sequence of measures is strictly decreasing

$$\mu(M) > \mu(M') > \cdots$$

Hence, there is a finite number of terms in the reduction path, ending in a normal form N.

$$\mu(M) > \mu(M') > \cdots > \mu(N)$$

□

Strong normalization of $\lambda\!\to$

A typed calculus has strong normalization if all legal terms are strongly normalizing.

We will prove that $\lambda\!\to$ *à la Curry* (without type annotations in lambdas) has the strong normalization property.

We will use a formalization of $\mathsf{SN}(\mathsf{M})$ as a first order formula based on a convenient characterization: the existence of a **bound** (in length) $\nu(\mathsf{M})$ for all reduction paths starting in M.

This characterization uses the following classic result of graph theory:

Lemma: (König's lemma) a finitely branching tree with no infinite path from the root is finite.

Formalization of strongly normalizing term

Lemma: for all $M \in \Lambda_{\rightarrow}$, we have $\mathsf{SN}(M)$ if, and only if, $\exists n \in \mathbb{N}$ such that all reduction paths starting in M have length at most n.

Notation: we write $\nu(M) = n$ to say that n is the bound of term M, and $\mathsf{SN}_n(M)$ to indicate that M is strongly normalizing with bound $\nu(M) = n$.

Proof:

- $(\Rightarrow)$ assume $\mathsf{SN}(M)$. Then, we can build a tree of reductions based on the available redexes at each moment. This tree is finitely branching, since each term has a finite number of redexes. Since $\mathsf{SN}(M)$, there is no infinite path from the root. By König's lemma this tree is finite, and therefore its height can be used as the bound $\nu(M)$.
- $(\Leftarrow)$ is direct, since it implies that all paths are finite.

Therefore, we define $\mathsf{SN}(M) \overset{\text{def}}{=} \exists \nu \in \mathbb{N},\ \forall M',\ M \twoheadrightarrow_\beta M' \Rightarrow |M \twoheadrightarrow_\beta M'| \leq \nu$

Lemma: if $\mathsf{SN}(M)$ and $M \rightarrow_\beta M'$, then $\nu(M') < \nu(M)$. Proof: direct.

Strong normalization of subterms

Lemma: **(strong normalization of subterm)** if $\mathsf{SN}(M)$ and N is a subterm of M, then $\mathsf{SN}(N)$.

Notation: let us denote $M = \mathcal{C}[N]$, where $\mathcal{C}$ represents the context around term N.

Proof: assume (for contradiction) that N is not strongly normalizing. Hence there is an infinite reduction path starting in N.

$$N \to_\beta N_1 \to_\beta N_2 \to_\beta \cdots$$

Due to the compatibility rules of beta reduction, this leads to the existence of an infinite reduction

$$\mathcal{C}[N] \to_\beta \mathcal{C}[N_1] \to_\beta \mathcal{C}[N_2] \to_\beta \cdots$$

which would lead to $M = \mathcal{C}[N]$ not being strongly normalizing. Contradiction!

Therefore, $\mathsf{SN}(N)$. $\square$

Logical relations (Tait's method)

Logical Relations is a proof method useful to prove many meta-properties of typed lambda calculi (but also of programming languages in general).

Roughly speaking, it can be described as the combination of

- constructing of a family of predicates indexed by types, with a *semantic* interpretation of function types.
- handling *open terms* (under a particular type environment) by means of *substitutions that conform to the type environment*

These two aspects will become clear by means of a concrete example: we will prove the property of *strong normalization* for STLC (à la Curry) using *logical relations.*

This technique was introduced by Tait (in *Intensional Interpretations of Functionals of Finite Type I*, 1967), and it is also known as *Tait's method.* This particular proof is due to Girard (from *Proofs and Types, 1989*) with minor adjustments.

Reducibility at a type

We define the family of predicates $\mathcal{R}_\tau$ by induction on type τ. We read $M \in \mathcal{R}_\tau$ as M is reducible at type τ.

$$\begin{aligned}
\mathcal{R}_X &\overset{\text{def}}{=} \{\, M \mid SN(M) \,\} \\
\mathcal{R}_{\tau_1 \to \tau_2} &\overset{\text{def}}{=} \{\, M \mid \forall N \in \Lambda,\ (N \in \mathcal{R}_{\tau_1}) \Rightarrow (M\,N \in \mathcal{R}_{\tau_2}) \,\}
\end{aligned}$$

Notice:

- this notion of *reducibility at* τ is **abstract** and does not have any direct relation to beta reduction or the presence of redexes (the name is misleading).
- for any atomic type X, the elements of $\mathcal{R}_X$ consist of all strongly normalizing terms, **independent if they are well-typed or not**. $M \in \mathcal{R}_\tau$ **does not imply** that $\vdash M : \tau$. For example, $(\lambda x.x\,x) \in \mathcal{R}_X$ and $x \in \mathcal{R}_X$.
- $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ means the following: *whenever* M *receives a reducible argument* N *at type* τ_1*, the application* $M\,N$ *is a reducible* term at type τ_2.
- $\mathcal{R}_{\tau_1 \to \tau_2}$ is defined in terms of $\mathcal{R}_{\tau_1}$ and $\mathcal{R}_{\tau_2}$ (i.e. this family of predicates is well-founded).

Properties of the reducibility predicate

Definition: a term M is **neutral** if it is **not** a lambda abstraction, i.e. $M \equiv x$ or $M \equiv M_1\ M_2$.

The following three properties of $\mathcal{R}_\tau$ will be essential for the proof of strong normalization.

Theorem: For any type τ,

1. (sn) if $M \in \mathcal{R}_\tau$ then $SN(M)$
2. (fw) if $M \in \mathcal{R}_\tau$ and $M \to_\beta M'$ then $M' \in \mathcal{R}_\tau$
3. (bw) if M is neutral and $(\forall M', M \to_\beta M' \Rightarrow M' \in \mathcal{R}_\tau)$, then $M \in \mathcal{R}_\tau$

A corollary of (3) is the following: if M is a *neutral normal form* (for example, x or $x\ y$), then $M \in \mathcal{R}_\tau$ for any type τ.

Proof: of all three simultaneously by induction on the structure of type τ.

Properties of reducibility, case $\tau \equiv X$

Show:

1. (sn) if $M \in \mathcal{R}_X$ then $SN(M)$
2. (fw) if $M \in \mathcal{R}_X$ and $M \to_\beta M'$ then $M' \in \mathcal{R}_X$
3. (bw) if M is neutral and $(\forall M', M \to_\beta M' \Rightarrow M' \in \mathcal{R}_X)$, then $M \in \mathcal{R}_X$

Proof:

1. (sn) direct from definition
2. (fw) from (sn) we have $SN(M)$, which means there is $\nu(M) \in \mathbb{N}$. Since $M \to M'$, we have $\nu(M') < \nu(M)$, which allows us to conclude $SN(M')$.
3. (bw) if M is normal form, $\nu(M) = 0$ and $SN(M)$. If there are M' such that $M \to_\beta M'$, consider η to be the maximum of all $\nu(M')$. From $\nu(M) = 1 + \eta$ we have $SN(M)$.

Properties of reducibility, case $\tau \equiv \tau_1 \to \tau_2$ (sn)

We can use (sn), (fw) and (bw) at types τ_1 and τ_2 as inductive hypotheses.

(sn) if $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ then $SN(M)$

Proof: (sn)

1. x is a neutral normal form. From (bw) at type τ_1, we have $x \in \mathcal{R}_{\tau_1}$
2. from $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ and $x \in \mathcal{R}_{\tau_1}$ we have $(M\ x) \in \mathcal{R}_{\tau_2}$
3. from (sn) at type τ_2, we have $SN(M\ x)$.
4. from (3) we obtain $SN(M)$ by strong normalization of subterm.

Properties of reducibility, case $\tau \equiv \tau_1 \to \tau_2$ (fw)

We can use (sn), (fw) and (bw) at types τ_1 and τ_2 as inductive hypotheses.

(fw) if $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ and $M \to_\beta M'$ then $M' \in \mathcal{R}_{\tau_1 \to \tau_2}$

Proof: (fw)

1. $M' \in \mathcal{R}_{\tau_1 \to \tau_2}$ means $\forall U,\ U \in \mathcal{R}_{\tau_1} \Rightarrow M'\ U \in \mathcal{R}_{\tau_2}$
2. Assume $U \in \mathcal{R}_{\tau_1}$.
   - 2.1 from $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ and $U \in \mathcal{R}_{\tau_1}$ we have $(M\ U) \in \mathcal{R}_{\tau_2}$
   - 2.2 from $M \to_\beta M'$ we have $(M\ U) \to_\beta (M'\ U)$
   - 2.3 from (fw) at type τ_2, (2.1) and (2.2) we obtain $(M'\ U) \in \mathcal{R}_{\tau_2}$, as required.
3. Hence, $M' \in \mathcal{R}_{\tau_1 \to \tau_2}$.

Properties of reducibility, case $\tau \equiv \tau_1 \to \tau_2$ (bw)

We can use (sn), (fw) and (bw) at types τ_1 and τ_2 as inductive hypotheses.

(bw) if M is neutral and $(\forall M', M \to_\beta M' \Rightarrow M' \in \mathcal{R}_{\tau_1 \to \tau_2})$, then $M \in \mathcal{R}_{\tau_1 \to \tau_2}$

Proof: (bw) $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ means $\forall U,\ U \in \mathcal{R}_{\tau_1} \Rightarrow M\ U \in \mathcal{R}_{\tau_2}$
Assume $U \in \mathcal{R}_{\tau_1}$.

1. from (sn) at τ_1, we have $SN(U)$ and therefore there is a number $\nu(U) \in \mathbb{N}$.
2. we reason by induction on $\nu(U)$ to show that any N such that $(M\ U) \to_\beta N$ is in $\mathcal{R}_{\tau_2}$. The term $(M\ U)$ may reduce to:
   - $N \equiv (M'\ U)$, with $M \to_\beta M'$. From the premisse we obtain $M' \in \mathcal{R}_{\tau_1 \to \tau_2}$, which entails $(M'\ U) \in \mathcal{R}_{\tau_2}$.
   - $N \equiv (M\ U')$, with $U \to U'$. Notice that $\nu(U') < \nu(U)$, and we obtain $(M\ U') \in \mathcal{R}_{\tau_2}$ from the inductive hypothesis.
3. since $(M\ U)$ is neutral and $(\forall N, (M\ U) \to N \Rightarrow N \in \mathcal{R}_{\tau_2})$, we have $(M\ U) \in \mathcal{R}_{\tau_2}$ by (bw) at τ_2.

Since we have concluded all cases, (sn), (fw) and (bw) hold at any type. $\square$

Abstraction lemma

Lemma: if $(\forall U,\ U \in \mathcal{R}_{\tau_1} \ \Rightarrow\ [U/x]M \in \mathcal{R}_{\tau_2})$ then $(\lambda x.M) \in \mathcal{R}_{\tau_1 \to \tau_2}$.

Proof: $(\lambda x.M) \in \mathcal{R}_{\tau_1 \to \tau_2}$ means $(\forall U,\ U \in \mathcal{R}_{\tau_1} \ \Rightarrow\ (\lambda x.M)\ U \in \mathcal{R}_{\tau_2})$

1. assume U such that $U \in \mathcal{R}_{\tau_1}$.
2. recall that $x \in \mathcal{R}_{\tau_1}$ and $[x/x]M = M$. From the premisse, we have $M \in \mathcal{R}_{\tau_2}$
3. we reason by induction on $\nu(M) + \nu(U)$ to show that every N such that $(\lambda x.M)\ U \to_\beta N$ is in $\mathcal{R}_{\tau_2}$. The term $(\lambda x.M)\ U$ may reduce to
   - $N \equiv [U/x]M$. We have $[U/x]M \in \mathcal{R}_{\tau_2}$ from the premisse.
   - $N \equiv (\lambda x.M')\ U$ with $M \to M'$. Notice that $\nu(M') < \nu(M)$, and we have $(\lambda x.M')\ U \in \mathcal{R}_{\tau_2}$ from the inductive hypothesis
   - $N \equiv (\lambda x.M)\ U'$ with $U \to U'$. Notice that $\nu(U') < \nu(U)$, and we have $(\lambda x.M)\ U' \in \mathcal{R}_{\tau_2}$ from the inductive hypothesis
4. Since $(\lambda x.M)\ U$ is neutral and for all N such that $(\lambda x.M)\ U \to_\beta N$ we have $N \in \mathcal{R}_{\tau_2}$, by (bw) at type τ_2 we obtain $(\lambda x.M)\ U \in \mathcal{R}_{\tau_2}$.
5. (1-4) suffices to show that $(\lambda x.M) \in \mathcal{R}_{\tau_1 \to \tau_2}$.

Reducibility under contexts

We now define a family of predicates $\mathcal{R}_{\Gamma,\tau}$ indexed by a pair (Γ, τ).

A term M is reducible at $(\bullet, \tau)$ if it is reducible at τ.

A term M is reducible at $((\Gamma, x : \tau_1), \tau)$ when the result of substituting x in M by a term U, reducible at (Γ, τ_1), always results in a term $[U/x]M$ reducible at (Γ, τ).

$$\begin{array}{rcl} \mathcal{R}_{\bullet,\tau} & \overset{\text{def}}{=} & \{\ M \ \mid\ M \in \mathcal{R}_{\tau}\ \} \\ \mathcal{R}_{(\Gamma,x:\tau_1),\tau} & \overset{\text{def}}{=} & \{\ M \ \mid\ \forall U, U \in \mathcal{R}_{\Gamma,\tau_1} \Rightarrow [U/x]M \in \mathcal{R}_{\Gamma,\tau}\ \} \end{array}$$

Notice:

- we assume that Γ can only be extended to $\Gamma, y : \tau'$ if $y \notin \mathrm{dom}(\Gamma)$.
- we write $\Gamma \vDash M : \tau$ as a synonym for $M \in \mathcal{R}_{\Gamma,\tau}$.
- reducibility at extended environment can be seen as a semantic version of the syntactic *substitution lemma* for $\lambda\!\rightarrow$.

Contextual lemmas for variables

We now will prove several lemmas related to reducibility under a given context, which we call *contextual lemmas*. All of them are easily proved by induction on the environment Γ, using the previous shown properties of $\mathcal{R}_\tau$ at the base case (when $\Gamma \equiv \bullet$).

Lemma: (ctx-var-1) if $\Gamma(x) = \tau$ then $\Gamma \vDash x : \tau$.
Proof: by induction on Γ.

- base case: $\Gamma, x{:}\tau \vDash x : \tau$, with $x \notin \mathsf{dom}(\Gamma)$. Assume U s.t. $\Gamma \vDash U : \tau$. We have $\Gamma \vDash [U/x]x : \tau$ from the assumption since $[U/x]x = U$.
- step: (IH) if $\Gamma(x) = \tau$ then $\Gamma \vDash x : \tau$.
  show: if $(\Gamma, y{:}\tau')(x) = \tau$, then $\Gamma, y{:}\tau' \vDash x : \tau$. This means assuming U s.t. $\Gamma \vDash U : \tau'$ and showing $\Gamma \vDash [U/y]x : \tau$. If $x = y$, then $[U/y]x = U$, $\tau' = \tau$ and also that $\Gamma \vDash [U/y]x : \tau$ holds from the premisse. If $x \neq y$, then $[U/y]x = x$ and $\Gamma \vDash [U/y]x : \tau$ holds by (IH).

Lemma: (ctx-var-2) if $x \notin \mathsf{dom}(\Gamma)$, then $\Gamma \vDash x : \tau$ (for any type τ).
Proof: by induction on Γ. base case: $x \in \mathcal{R}_\tau$ for any type τ. step: since $x \notin \mathsf{dom}(\Gamma)$, it will not be substituted and the property holds from the inductive hypothesis.

Contextual lemma for application

Lemma: (ctx-app)

$$\frac{\Gamma \vDash M : \tau_1 \to \tau_2 \qquad \Gamma \vDash N : \tau_1}{\Gamma \vDash M\ N : \tau_2}$$

Proof: by induction on Γ.

- base: from $M \in \mathcal{R}_{\tau_1 \to \tau_2}$ and $N \in \mathcal{R}_{\tau_1}$ we obtain $M\ N \in \mathcal{R}_{\tau_2}$ (direct from definition)
- step:
  (IH) for all M,N,τ_1, τ_2, $(\Gamma \vDash M : \tau_1 \to \tau_2) \Rightarrow (\Gamma \vDash N : \tau_1) \Rightarrow (\Gamma \vDash M\ N : \tau_2)$
  Assume
  - $(\Gamma, y{:}\tau' \vDash M : \tau_1 \to \tau_2)$
  - $(\Gamma, y{:}\tau' \vDash N : \tau_1)$

  and show $(\Gamma, y{:}\tau' \vDash M\ N : \tau_2)$.
  General steps: assume U s.t. $\Gamma \vDash U : \tau_1$, propagate $[U/x]$ over $(M\ N)$ and use the inductive hypothesis on $[U/x]M$ and $[U/x]N$ to obtain the result.

Contextual lemma for lambda

Lemma: (ctx-lam)

$$\frac{\Gamma, x : \tau_1 \vDash M : \tau_2}{\Gamma \vDash (\lambda x.M) : \tau_1 \to \tau_2}$$

Proof: by induction on Γ.

- base: direct use of the abstraction lemma.
- step: straightforward application of inductive hypothesis.

Contextual lemma for (sn)

Lemma: (ctx-sn) if $\Gamma \vDash M : \tau$ then $\mathsf{SN}(M)$

Proof: by induction on Γ.

- base: direct from (sn) at type τ
- step: (IH)   (for all M, τ) if $\Gamma \vDash M : \tau$ then $\mathsf{SN}(M)$
  1. assume $\Gamma, y{:}\tau' \vDash M : \tau$. We must show $\mathsf{SN}(M)$
  2. notice that $y \notin \mathrm{dom}(\Gamma)$, hence we have $\Gamma \vDash y : \tau'$ by (ctx-var-2).
  3. from (1) and (2) we obtain $\Gamma \vDash [y/y]M : \tau$
  4. since $[y/y]M = M$, from (3) we have $\Gamma \vDash M : \tau$
  5. from (4) and (IH), we obtain $\mathsf{SN}(M)$

Strong normalization of $\lambda\!\rightarrow$ (proof)

Now we can finally prove the main theorem.

Theorem: for all M, if there are Γ and τ such that $\Gamma \vdash \mathsf{M} : \tau$, then $\mathsf{SN}(\mathsf{M})$

Proof: the main result is obtained in two steps:

- (a) if $\Gamma \vdash \mathsf{M} : \tau$ then $\Gamma \vDash \mathsf{M} : \tau$
- (b) if $\Gamma \vDash \mathsf{M} : \tau$ then $\mathsf{SN}(\mathsf{M})$

(a) is proved by induction on the structure of the type derivation $\Gamma \vdash \mathsf{M} : \tau$, using the contextual lemmas for variables, lambdas and applications.

(b) is (ctx-sn), which we proved by induction on Γ. $\square$

Content

Extensions of $\lambda\!\rightarrow$

Extensions of $\lambda_{\rightarrow}$

We now will consider some extensions of $\lambda_{\rightarrow}$:

- empty type
- union type
- product types
- disjoint union types
- primitive types (nat, bool)
- typed fixpoint operators

The combination of these feature (and others) create several variations of $\lambda_{\rightarrow}$.

Empty type

We can introduce an **empty type** (named 0) in the type language.

This type is special because there is no closed term M such that $\bullet \vdash M : 0$.

However, we can declare variables of such type in the environment, and use it in function parameters. Although it may seem counter-intuitive, this type will be quite useful in the future (when discussing the Curry-Howard correspondence)

Syntax of types:

$$\tau ::= \cdots \mid 0$$

Syntax of terms: (unchanged)

Type system: (unchanged)

Reduction: (unchanged)

Unit type

We can also introduce a **unit type** (named 1) which has a single value $\langle\rangle$ (pronounced also unit)

The value $\langle\rangle$ is an atomic normal form whose introduction does not affect the reduction relation.

Syntax of types:

$$\tau \ ::= \ \cdots \ | \ 1$$

Syntax of terms:

$$M \ ::= \ \cdots \ | \ \langle\rangle$$

Type system:

$$\frac{}{\Gamma \vdash \langle\rangle : 1} \quad (\textsc{unit})$$

Reduction: (unchanged)

Empty type (revisited)

A better version of an empty type 0 comes with a single destructor, which we will call bot.

When provided with an argument of type 0, the expression $\mathsf{bot}\ M$ has any possible type. This behaviour is interesting because it reproduces the $\bot - \mathsf{elim}$ rule of propositional logic.

Syntax of types: $\tau \ ::= \ \cdots \ | \ 0$

Syntax of terms: $M ::= \cdots \ | \ \mathsf{bot}\ M'$

Type system:

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \mathsf{bot}\ M : \tau} \quad (\textsc{bot})$$

Reduction:

$$\frac{M \longrightarrow_\beta M'}{\mathsf{bot}\ M \longrightarrow_\beta \mathsf{bot}\ M'} \quad (\textsc{bot-arg})$$

Notice: Notice that 1 has a single constructor but no destructor, the actual inverse of this version of 0.

Product types: syntax

A product type $\tau_1 \times \tau_2$ corresponds to the collection of ordered pairs $\langle a, b \rangle$ in which a is of type τ_1 and b is of type τ_2.

The term $\langle \_, \_ \rangle$ is an infixed *constructor* with two parameters.

We also need to introduce *destructors* (projections) fst and snd to extract the pair components.

Syntax of types:

$$\tau ::= \cdots \mid \tau_1 \times \tau_2$$

Syntax of terms:

$$M ::= \cdots \mid \langle M_1, M_2 \rangle \mid \mathsf{fst}\ M_1 \mid \mathsf{snd}\ M_1$$

Product types: type system

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash \langle M_1, M_2 \rangle : \tau_1 \times \tau_2} \quad (\textsc{pair})$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ M : \tau_1} \quad (\textsc{fst})$$

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ M : \tau_2} \quad (\textsc{snd})$$

Product types: reduction

$$\frac{M_1 \longrightarrow_\beta M_1'}{\langle M_1, M_2\rangle \longrightarrow_\beta \langle M_1', M_2\rangle}\ (\textsc{pair-left})$$

$$\frac{}{\mathsf{fst}\ \langle M_1, M_2\rangle \longrightarrow_\beta M_1}\ (\textsc{fst-pair})$$

$$\frac{M_2 \longrightarrow_\beta M_2'}{\langle M_1, M_2\rangle \longrightarrow_\beta \langle M_1, M_2'\rangle}\ (\textsc{pair-right})$$

$$\frac{M \longrightarrow_\beta M'}{\mathsf{snd}\ M \longrightarrow_\beta \mathsf{snd}\ M'}\ (\textsc{snd-arg})$$

$$\frac{M \longrightarrow_\beta M'}{\mathsf{fst}\ M \longrightarrow_\beta \mathsf{fst}\ M'}\ (\textsc{fst-arg})$$

$$\frac{}{\mathsf{snd}\ \langle M_1, M_2\rangle \longrightarrow_\beta M_2}\ (\textsc{snd-pair})$$

Disjoint union types: syntax

A disjoint union type $\tau_1 + \tau_2$ corresponds to a type which elements come from τ_1 (prefixed by left) or come from τ_2 (prefixed by right).

Both left and right are constructors of one parameter. We also have a ternary $\mathsf{case}\ M[x \mapsto M_1, y \mapsto M_2]$ destructor, which returns M_1 or M_2 dependeding of M being a left or a right constructor. The argument of the left (resp. right) constructor is bound to x (resp. y) and can be used in M_1 (resp. M_2).

Syntax of types:

$$\tau \ ::= \ \cdots \ \mid \ \tau_1 + \tau_2$$

Syntax of terms:

$$M \ ::= \ \cdots \ \mid \ \mathsf{left}\ M_1 \ \mid \ \mathsf{right}\ M_1 \ \mid \ \mathsf{case}\ M_1\ [x \mapsto M_2, y \mapsto M_3]$$

Disjoin union types: type system

$$\frac{\Gamma \vdash \mathsf{M} : \tau_1}{\Gamma \vdash \mathsf{left}\ \mathsf{M} : \tau_1 + \tau_2} \quad (\textsc{left})$$

$$\frac{\Gamma \vdash \mathsf{M} : \tau_2}{\Gamma \vdash \mathsf{right}\ \mathsf{M} : \tau_1 + \tau_2} \quad (\textsc{right})$$

$$\frac{\Gamma \vdash \mathsf{M} : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash \mathsf{M}_1 : \tau' \qquad \Gamma, y : \tau_2 \vdash \mathsf{M}_2 : \tau'}{\Gamma \vdash \mathsf{case}\ \mathsf{M}\ [x \mapsto \mathsf{M}_1, y \mapsto \mathsf{M}_2] : \tau'} \quad (\textsc{case})$$

Disjoint union types: reduction

$$\frac{M \longrightarrow_\beta M'}{\mathsf{case}\ M\ [x \mapsto M_1, y \mapsto M_2] \longrightarrow_\beta \mathsf{case}\ M'\ [x \mapsto M_1, y \mapsto M_2]}\ (\textsc{case-arg1})$$

$$\frac{M_1 \longrightarrow_\beta M_1'}{\mathsf{case}\ M\ [x \mapsto M_1, y \mapsto M_2] \longrightarrow_\beta \mathsf{case}\ M\ [x \mapsto M_1', y \mapsto M_2]}\ (\textsc{case-arg2})$$

$$\frac{M_2 \longrightarrow_\beta M_2'}{\mathsf{case}\ M\ [x \mapsto M_1, y \mapsto M_2] \longrightarrow_\beta \mathsf{case}\ M\ [x \mapsto M_1, y \mapsto M_2']}\ (\textsc{case-arg3})$$

continues...

Disjoint union types: reduction (cont.)

$$\frac{}{\mathsf{case}\ (\mathsf{left}\ M)\ [x \mapsto M_1, y \mapsto M_2] \longrightarrow_\beta [M/x]M_1} \quad (\textsc{case-left})$$

$$\frac{}{\mathsf{case}\ (\mathsf{right}\ M)\ [x \mapsto M_1, y \mapsto M_2] \longrightarrow_\beta [M/y]M_2} \quad (\textsc{case-right})$$

$$\frac{M \longrightarrow_\beta M'}{\mathsf{left}\ M \longrightarrow_\beta \mathsf{left}\ M'} \quad (\textsc{left-arg})$$

$$\frac{M \longrightarrow_\beta M'}{\mathsf{right}\ M \longrightarrow_\beta \mathsf{right}\ M'} \quad (\textsc{right-arg})$$

Gödel's System T

We can identify two ages/categories for Type theory:

- Early type theories (Russel, Church). From the beginning of 20th century to Church's paper in 1940
- Modern type theories (de Bruijn, Martin Löf, Coquand, Huet ...). From the first version of Automath, in 1967

Gödel's System T (1958) is at the transition from early to modern type theories

It can be described as and extension of $\lambda\rightarrow$ with product types and two primitive base types: nat and bool.

Gödel's System T - Syntax

$$
\begin{array}{lcl}
\tau & ::= & \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \mathtt{nat} \mid \mathtt{bool} \\
M & ::= & x \mid \lambda x : \tau . M \mid M\ N \mid \langle M_1, M_2 \rangle \mid \mathsf{fst}\ M_1 \mid \mathsf{snd}\ M_1 \\
 & & \mathbf{0} \mid \mathbf{S}\ M_1 \mid \mathbf{R}\ M_1\ M_2\ M_3 \mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{D}\ M_1\ M_2\ M_3
\end{array}
$$

Intended meaning:

- $\mathbf{0}$ is the usual constant and $\mathbf{S}\ M$ means $M + 1$.
- $\mathbf{R}$ is an operator for primitive recursion on natural number:
  - $\mathbf{R}\ M\ N\ \mathbf{0} = M \qquad \mathbf{R}\ M\ N\ (\mathbf{S}\ Q) = N\ (\mathbf{R}\ M\ N\ Q)\ Q$
- $\mathbf{T}$ and $\mathbf{F}$ are the truth values
- $\mathbf{D}$ is the conditional operator (if-then-else), but with the boolean to be tested as the last argument
  - $\mathbf{D}\ M\ N\ \mathbf{T} = M \qquad \mathbf{D}\ M\ N\ \mathbf{F} = N$

Gödel's System T - Reduction Rules

The redexes are **the same** as those of (un)typed lambda calculus plus:

$$\mathbf{R}\ M\ N\ \mathbf{0} \longrightarrow_\beta M \qquad \mathbf{D}\ M\ N\ \mathbf{T} \longrightarrow_\beta M$$

$$\mathbf{R}\ M\ N\ (\mathbf{S}\ Q) \longrightarrow_\beta N\ (\mathbf{R}\ M\ N\ Q)\ Q \qquad \mathbf{D}\ M\ N\ \mathbf{F} \longrightarrow_\beta N$$

together with rules for evaluating the arguments of constructors. Considering **R** we have the following three rules

$$\frac{M \longrightarrow_\beta M'}{\mathbf{R}\ M\ N\ P \longrightarrow_\beta \mathbf{R}\ M'\ N\ P} \qquad \frac{N \longrightarrow_\beta N'}{\mathbf{R}\ M\ N\ P \longrightarrow_\beta \mathbf{R}\ M\ N'\ P} \qquad \frac{P \longrightarrow_\beta P'}{\mathbf{R}\ M\ N\ P \longrightarrow_\beta \mathbf{R}\ M\ N\ P'}$$

and there are similar rules for **S** and **D**.

Gödel's System T - Example of reduction

Example of a reduction sequence

$$
\begin{array}{lll}
\mathbf{R}\ \mathsf{M}\ \mathsf{N}\ 4 & \longrightarrow & \mathsf{N}\ (\mathbf{R}\ \mathsf{M}\ \mathsf{N}\ 3)\ 3 \\
 & \longrightarrow & \mathsf{N}\ (\mathsf{N}\ (\mathbf{R}\ \mathsf{M}\ \mathsf{N}\ 2)\ 2)\ 3 \\
 & \longrightarrow & \mathsf{N}\ (\mathsf{N}\ (\mathsf{N}\ (\mathbf{R}\ \mathsf{M}\ \mathsf{N}\ 1)\ 1)\ 2)\ 3 \\
 & \longrightarrow & \mathsf{N}\ (\mathsf{N}\ (\mathsf{N}\ (\mathsf{N}\ (\mathbf{R}\ \mathsf{M}\ \mathsf{N}\ 0)\ 0)\ 1)\ 2)\ 3 \\
 & \longrightarrow & \mathsf{N}\ (\mathsf{N}\ (\mathsf{N}\ (\mathsf{N}\ \mathsf{M}\ 0)\ 1)\ 2)\ 3
\end{array}
$$

Gödel's System T - Type System

All the typing rules for $\lambda\!\rightarrow$ plus the rules for constants $\mathbf{0}$, $\mathbf{T}$, and $\mathbf{F}$:

$$\Gamma \vdash \mathbf{0} : \mathsf{nat} \ (\text{ZERO}) \qquad \Gamma \vdash \mathbf{T} : \mathtt{bool} \ (\text{TRUE}) \qquad \Gamma \vdash \mathbf{F} : \mathtt{bool} \ (\text{FALSE})$$

and typing rules for $\mathbf{S}$, $\mathbf{D}$, and $\mathbf{R}$:

$$\frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathbf{S}\ M : \mathsf{nat}} \quad (\text{SUCC})$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau \qquad \Gamma \vdash P : \mathtt{bool}}{\Gamma \vdash \mathbf{D}\ M\ N\ P : \tau} \quad (\text{COND})$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau \rightarrow \mathsf{nat} \rightarrow \tau \qquad \Gamma \vdash P : \mathsf{nat}}{\Gamma \vdash \mathbf{R}\ M\ N\ P : \tau} \quad (\text{REC})$$

Gödel's System T - examples (I)

Logical operators:

$\mathtt{neg}(\mathsf{M}) = \mathbf{D}\,\mathbf{F}\,\mathbf{T}\,\mathsf{M} \qquad \mathtt{disj}(\mathsf{M}, \mathsf{N}) = \mathbf{D}\,\mathbf{T}\,\mathsf{N}\,\mathsf{M} \qquad \mathtt{conj}(\mathsf{M}, \mathsf{N}) = \mathbf{D}\,\mathsf{N}\,\mathbf{F}\,\mathsf{M}$

So we have that

$$\mathtt{disj}(\mathbf{T}, \mathsf{N}) \twoheadrightarrow_\beta \mathbf{T} \qquad \mathtt{disj}(\mathbf{F}, \mathsf{N}) \twoheadrightarrow_\beta \mathsf{N} \qquad \mathtt{disj}(\mathsf{N}, \mathbf{F}) \twoheadrightarrow_\beta \mathsf{N}$$

Gödel's System T - examples (II)

We know that $\mathsf{plus}(x, \mathbf{0}) = x$ and $\mathsf{plus}(x, \mathbf{S}(y)) = \mathbf{S}(\mathsf{plus}(x, y))$.

In System T:

$$\mathsf{plux}(x, y) = \mathbf{R}\ x\ (\lambda z{:}\mathtt{nat}.\ \lambda z'{:}\mathtt{nat}.\ \mathbf{S}(z))\ \ y$$

High-order iterator function $\mathtt{it}$ such that $\mathsf{it}\ (f)\ \overline{n} = f^n\ \overline{1}$:

$$\mathsf{it}\ (f)\ x = \mathbf{R}\ \overline{1}\ (\lambda z{:}\mathtt{nat}.\ \lambda z'{:}\mathtt{nat}.\ f\ (z))\ x$$

One could also introduce an iterator constant $\mathbf{It}$ in the language:

$$\mathbf{It}\ \ M\ N\ (\mathbf{S}\ P)\ \ \longrightarrow_\beta\ \ N\ (\mathbf{It}\ M\ N\ P)$$

System T satisfies Church-Rosser and strong normalization: all (maximal) reduction sequences end with a normal form. Hence, all functions defined with System T are total.

All total functions of Peano Arithmetic can be defined with system T.

Programming Language for Computable Functions

LCF, a Logic for Computable Functions, was introduced by Dana Scott in a influential and widely circulated manuscript from 1969, later published in 1993

Robin Milner, who named the logic of Scott as LCF, developed a theorem prover for it called Stanford LCF (1972). In 1973 he developed Edimburgh LCF

The metalanguage of Edimburgh LCF evolved to become the general programming language ML (for Meta Language)

In 1977, Gordon Plotkin published the paper ***LCF Considered as a Programming Language***. He took the terms from the logic LCF to make his programming language **PCF**.

PCF - Syntax

PCF can be seen as core of modern typed functional programming languages, such as Haskell, OCaml

$$
\begin{array}{lcl}
\tau & ::= & \texttt{nat} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \\
\\
M & ::= & x \mid \lambda x : \tau.\, M \mid M\, N \\
 & & 0 \mid T \mid F \mid \texttt{succ}\, M \mid \texttt{pred}\, M \mid \texttt{if}\, M\, \texttt{then}\, M_1\, \texttt{else}\, M_2 \mid \texttt{iszero}\, M \\
 & & \texttt{fix}\, M
\end{array}
$$

PCF terms are not strong normalising: given a well-typed term of PCF its evaluation might not terminate

That happen because of the introduction of a fixed point operator in the language to support general recursion

PCF - Type System and Reduction Rules

The typing rules of PCF are all as expected. The typing rule for $\texttt{fix}\ \mathsf{M}$ is:

$$\frac{\Gamma \vdash \mathsf{M} : \tau \to \tau}{\Gamma \vdash \texttt{fix}\ \mathsf{M} : \tau}$$

The original definition of PCF received a *call by name* operational semantics. Here we will define a *call by value* semantics

The interesting rules are those for $\texttt{fix}\ \mathsf{M}$:

$$\frac{\mathsf{M} \quad \longrightarrow_\beta \quad \mathsf{M}'}{\texttt{fix}\ \mathsf{M} \quad \longrightarrow_\beta \quad \texttt{fix}\ \mathsf{M}'}$$

$$\texttt{fix}\ \ \lambda x : \tau.\ \mathsf{M} \quad \longrightarrow_\beta \quad [\texttt{fix}\ \lambda x : \tau.\ \mathsf{M}/x]\ \mathsf{M}$$

Examples (I)

We can now write a program in PCF to test if $\mathsf{iseven}\ 5$, for instance:

$$
\begin{array}{lcl}
\mathsf{ff} & = & \lambda\, \mathsf{isev} : \mathtt{nat} \to \mathtt{bool} \\
 & & \quad \lambda\, \mathsf{x} : \mathtt{nat} \\
 & & \quad\quad \mathtt{if}\ (\mathtt{iszero}\ \mathsf{x})\ \mathtt{then}\ \mathtt{true} \\
 & & \quad\quad \mathtt{else}\ \mathtt{if}\ (\mathtt{iszero}\ (\mathtt{pred}\ \mathsf{x}))\ \mathtt{then}\ \mathtt{false} \\
 & & \quad\quad \mathtt{else}\ \mathsf{isev}\ (\mathtt{pred}\ (\mathtt{pred}\ \mathsf{x})) \\
 & & \\
\mathsf{iseven} & = & \mathtt{fix}\ \mathtt{ff}
\end{array}
$$

Examples (II)

A PCF program to add two natural numbers (plus 2 3, for instance)

$$
\begin{aligned}
\mathsf{ff} \quad &= \quad \lambda\, \mathsf{pl} : \mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat})\\
&\qquad \lambda\, \mathsf{x} : \mathsf{nat}.\, \lambda\, \mathsf{y} : \mathsf{nat}\,.\\
&\qquad\quad \mathsf{if}\ (\mathsf{iszero}\ \mathsf{y})\ \mathsf{then}\ \mathsf{x}\\
&\qquad\quad \mathsf{else}\ \mathsf{succ}\ (\mathsf{pl}\ \mathsf{x}\ (\mathsf{pred}\ \mathsf{y}))\\
\\
\mathsf{plus} \quad &= \quad \mathsf{fix}\ \mathsf{ff}
\end{aligned}
$$

Properties (I)

A term M is in normal form if there is no M' such that $\mathsf{M} \longrightarrow \mathsf{M}'$.

The following normal forms are **values** of PCF

$$\begin{array}{lcl} \mathtt{n} & ::= & \mathtt{0} \mid \mathsf{S}\,\mathtt{n}' \\ \mathtt{v} & ::= & \mathtt{n} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda x{:}\tau.\,\mathsf{M} \end{array}$$

A normal form which is not a value is **stuck**. Example: `pred false`

The reduction sequence of a term M of PCF can either:

- diverge,
- end with a stuck term, or
- end with a value.

Properties (II)

A closed well type term is not stuck:

Theorem: (Progress) If $\varnothing \vdash M : \tau$ then (i) M is a value or (ii) there is M' such that $M \longrightarrow M'$

Types are preserved by one step reduction:

Theorem: (Preservation) If $\Gamma \vdash M : \tau$ and $M \longrightarrow M'$ then $\Gamma \vdash M' : \tau$

With Progress and Preservation we can prove *soundness of typing* w.r.t the operational semantics:

Theorem: If $\varnothing \vdash M : \tau$ then (i) $M \twoheadrightarrow v$ and $\varnothing \vdash v : \tau$, or (ii) $M \twoheadrightarrow M'$ implies $M' \rightarrow M''$ for some M''

PCF in PL research

Plotkin's original motivation was to study the relationship between denotational and operational semantics

PCF is widely used as a starting point for various investigations in language design

Overview - STLC $\lambda\rightarrow$

- atomic types, and function types (or arrow types)
- Church style vs Curry style
- terms such as self application and Y combinator are not legal terms in $\lambda\rightarrow$
- Church-Rosser
- Strong Normalization (proof based on *logical relations*)
- not Turing-Complete
- Subject Reduction (proof depends on a Substitution Lemma)
- Several possible extensions

Overview - System T

- $\lambda\rightarrow$ plus products, natural numbers (defined inductively with **0** and **S**), boolean constants, conditional (**D**) and (non-general) recursion **R**
- Church Rosser
- Strong normalization
- Subject Reduction (type preservation)
- More expressive than $\lambda\rightarrow$
- not Turing-Complete

Overview - PCF

- PCF = $\lambda\rightarrow$ plus booleans, natural numbers and general recursion ($\mathsf{fix}\ M$)
- it is **not** strong normalizing (because of general recursion)
- Turing-complete
- used the theoretical basis of functional programming languages with different evaluation strategies such as *call by value* and *call by name*
- type safety: “well-typed PCF programs do not go wrong/get stuck”
  - type preservation - one step reduction preserves types (subject reduction)
  - progress - well-typed terms, which are no values, can evolve one step

Content

Intuitionistic Propositional Logic

Isomorphsim of Curry-Howard

Syntax

We assume a countable infinite set $\mathcal{P}$ of **propositional symbols/letters** $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D} \ldots$

$\textsc{Prop}$, the set of **wff of propositional logic**, is the smallest set satisfying:

1. $\mathcal{P} \subseteq \textsc{Prop}$
2. $\bot, \top \in \textsc{Prop}$
3. if $\varphi \in \textsc{Prop}$ then$\neg\varphi \in \textsc{Prop}$
4. if $\varphi, \psi \in \textsc{Prop}$ then
    - $\varphi \wedge \psi \in \textsc{Prop}$
    - $\varphi \vee \psi \in \textsc{Prop}$
    - $\varphi \Rightarrow \psi \in \textsc{Prop}$

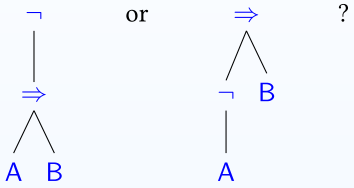
The set $\textsc{Prop}$ can also be defined by the following abstract grammar:

$$\varphi \quad ::= \quad \mathsf{A} \mid \bot \mid \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi$$

We write $\varphi \Leftrightarrow \psi$ for $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Syntax

Linear textual representation might introduce ambiguity. Ex: Which of these trees corresponds to $\neg A \Rightarrow B$?



If in doubt, use parenthesis: $\neg(A \Rightarrow B)$ vs $(\neg A) \Rightarrow B$

Syntax

Convention to reduce number of parenthesis:

1. Conjunction and disjunction are left associative. Ex:

$$\begin{aligned} A \wedge B \wedge C &= ((A \wedge B) \wedge C) \\ A \vee B \vee C &= ((A \vee B) \vee C) \end{aligned}$$

2. Implication is right associative. Ex:

$$A \Rightarrow B \Rightarrow C = (A \Rightarrow (B \Rightarrow C))$$

3. Priority: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, in this order. Ex:

$$\begin{aligned} \neg A \wedge B &= (\neg A) \wedge B \\ A \vee B \wedge C &= A \vee (B \wedge C) \\ A \vee \neg B \Rightarrow C &= (A \vee (\neg B)) \Rightarrow C \end{aligned}$$

Semantics of Classical Propositional Logic

What is the truth value of these formulas :

$\mathsf{A} \wedge \mathsf{B}$ $\mathsf{A} \wedge \neg \mathsf{A}$ $\mathsf{A} \vee \neg \mathsf{A}$ $\mathsf{A} \Rightarrow (\mathsf{B} \Rightarrow (\mathsf{A} \wedge \mathsf{B}))$

Only for the first formula $\mathsf{A} \wedge \mathsf{B}$ the truth value depends of the truth value of the propositional letters.

In the other examples the truth-value is determined by the formula structure.

We assume that the valuation of the **propositional letters** is given by a function

$$\mathcal{V}_0 : \mathcal{P} \Rightarrow \{\mathsf{T}, \mathsf{F}\}$$

Semantics of Classical PL

The a valuation function $\mathcal{V}$ for formulas depends on the valuation function $\mathcal{V}_0$ for propositional symbols.

$$\mathcal{V} : \mathcal{L} \Rightarrow \{\mathsf{T}, \mathsf{F}\}$$

$\mathcal{V}(\varphi)$ is defined as follows:

Case $\varphi = \mathsf{A}$, for $\mathsf{A} \in \mathcal{P}$	$\mathcal{V}(\varphi) = \mathcal{V}_0(\mathsf{A})$		
Case $\varphi = \bot$	$\mathcal{V}(\varphi) = \mathsf{F}$		
Case $\varphi = \top$	$\mathcal{V}(\varphi) = \mathsf{T}$		
Case $\varphi = \neg\psi$	$\mathcal{V}(\varphi) = \mathsf{T}$	iff	$\mathcal{V}(\psi) = \mathsf{F}$
Case $\varphi = \psi \wedge \xi$	$\mathcal{V}(\varphi) = \mathsf{T}$	iff	$\mathcal{V}(\psi) = \mathsf{T}$ and $\mathcal{V}(\xi) = \mathsf{T}$
Case $\varphi = \psi \vee \xi$	$\mathcal{V}(\varphi) = \mathsf{T}$	iff	$\mathcal{V}(\psi) = \mathsf{T}$ or $\mathcal{V}(\xi) = \mathsf{T}$
Case $\varphi = \psi \Rightarrow \xi$	$\mathcal{V}(\varphi) = \mathsf{T}$	iff	$\mathcal{V}(\psi) = \mathsf{F}$ or $\mathcal{V}(\xi) = \mathsf{T}$

We also write $\mathcal{V} \vDash \varphi$ to denote $\mathcal{V}(\varphi) = \mathsf{T}$.

Semantics for Classical PL

The truth value of formulas can be given in terms of **truth tables** for each connective:

φ	$\neg\varphi$
F	T
T	F

φ	ψ	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \Rightarrow \psi$
F	F	F	F	T
F	T	F	T	T
T	F	F	T	F
T	T	T	T	T

A **theory** Γ is a set of formulas. We say a valuation $\mathcal{V}$ satisfies a theory Γ, written $\mathcal{V} \vDash \Gamma$, when $\mathcal{V}(\varphi) = \mathsf{T}$ for all $\varphi \in \Gamma$.

We say that φ is a **logical consequence** of Γ, written $\Gamma \vDash \varphi$, when the following holds: for any valuation $\mathcal{V}$, if $\mathcal{V} \vDash \Gamma$ then $\mathcal{V} \vDash \varphi$.

Deductive Systems

A **deductive system** is a mathematical formalism with which we can obtain logical consequences from a theory Γ.

The notation $\Gamma \vdash_{\mathsf{X}} \varphi$, called a **sequent**, express that φ follows from the formulas in Γ according the deductive system X

Two deductive systems :

- **Axiomatic:** $\vdash_{\mathsf{AX}}$
- **Natural Deduction:** $\vdash_{\mathsf{ND}}$

We will focus on **Natural Deduction**.

Soundness and Completeness

Soundness and **Completeness** are properties of a *deductive system* $\vdash$ wrt *logical consequence* $\models$.

A deductive system $\vdash$ is **sound** iff

$$\text{if } \Gamma \vdash \varphi \text{ then } \Gamma \models \varphi$$

i.e., it never produces a formula φ which is not a logical consequence of Γ.

A deductive system $\vdash$ is **complete**, iff

$$\text{if } \Gamma \models \varphi \text{ then } \Gamma \vdash \varphi$$

i.e., it every logical consequence of Γ can be produced by it.

Laws of Classical Propositional Logic

Law	**Formula**
Double negation	$\varphi \Rightarrow \neg\neg\varphi$
Excluded middle	$\varphi \vee \neg\varphi$
Noncontradiction	$\neg(\varphi \wedge \neg\varphi)$
De Morgan	$\varphi \wedge \psi \Rightarrow \neg(\neg\varphi \vee \neg\psi)$
	$\varphi \vee \psi \Rightarrow \neg(\neg\varphi \wedge \neg\psi)$
Commutative	$\varphi \vee \psi \Rightarrow \psi \vee \varphi$
	$\varphi \wedge \psi \Rightarrow \psi \wedge \varphi$
Associative	$(\varphi \vee \psi) \vee \xi \Rightarrow \varphi \vee (\psi \vee \xi)$
	$(\varphi \wedge \psi) \vee \xi \Rightarrow \varphi \wedge (\psi \wedge \xi)$
Transposition	$\varphi \Rightarrow \psi \Rightarrow \neg\psi \Rightarrow \neg\varphi$
Distributive	$\varphi \wedge (\psi \vee \xi) \Rightarrow (\varphi \wedge \psi) \vee (\varphi \wedge \xi)$
	$\varphi \vee (\psi \wedge \xi) \Rightarrow (\varphi \vee \psi) \wedge (\varphi \vee \xi)$
Permutation	$\varphi \Rightarrow (\psi \Rightarrow \xi) \Rightarrow \psi \Rightarrow (\varphi \Rightarrow \xi)$
Sylogism	$(\varphi \Rightarrow \psi) \Rightarrow ((\varphi \Rightarrow \xi) \Rightarrow (\psi \Rightarrow \xi))$
Importation	$(\varphi \Rightarrow (\psi \Rightarrow \xi)) \Rightarrow ((\varphi \wedge \psi) \Rightarrow \xi)$
Exportation	$((\varphi \wedge \psi) \Rightarrow \xi) \Rightarrow (\varphi \Rightarrow (\psi \Rightarrow \xi))$

Natural Deduction

Natural deduction $\vdash_{\mathsf{DN}}$, proposed by *Gentzen*, is more intuitive than an axiomatic system.

Features:

- Each connective has *introduction* and *elimination rules*
- We can add and discharge/cancel hypotheses/assumptions to the deduction process

Obs 1: In the following, $\neg\varphi$ **is an abbreviation for** $\varphi \Rightarrow \bot$.

Obs 2: There are different styles to present rules and proofs in Natural Deduction.

Natural Deduction: conjunction

Introduction:

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi} \qquad (\wedge_{\mathsf{I}})$$

Elimination:

$$\frac{\varphi \wedge \psi}{\varphi} \qquad (\wedge_{\mathsf{E1}})$$

$$\frac{\varphi \wedge \psi}{\psi} \qquad (\wedge_{\mathsf{E2}})$$

Natural Deduction: disjunction

Introduction:

$$\frac{\varphi}{\varphi \vee \psi} \qquad (\vee_{I1})$$

$$\frac{\psi}{\varphi \vee \psi} \qquad (\vee_{I2})$$

Elimination:

$$\frac{\varphi \vee \psi \qquad \varphi \Rightarrow \xi \qquad \psi \Rightarrow \xi}{\xi} \qquad (\vee_{E})$$

Natural deduction: implication

Introduction:

$$\frac{\boxed{\begin{array}{l}\cancel{\varphi} \text{ (hypothesis)} \\ \quad\vdots \\ \psi \text{ (conclusion)}\end{array}}}{\varphi \Rightarrow \psi} \qquad (\Rightarrow_{\mathsf{I}})$$

Elimination (also know as *Modus Ponens*) :

$$\frac{\varphi \Rightarrow \psi \qquad \varphi}{\psi} \qquad (\Rightarrow_{\mathsf{E}})$$

Natural Deduction: contradiction

Two rules that "eliminate"contradiction :

Elimination - ex falso quodlibet

$$\frac{\bot}{\varphi} \qquad (\bot\text{E})$$

Elimination - reductio ad absurdum

$$\frac{\boxed{\begin{array}{c}\cancel{\neg\varphi}\\ \vdots \\ \bot\end{array}}}{\varphi} \qquad (\text{RAA})$$

RAA (Reductio ad absurdum) captures **proof by contradiction**.

Rule RAA is not present in intuitionistic (constructive) logic.

Example (I)

Prove the following sequent:

$$A, B \vdash (A \land B) \land A$$

Proof tree:

$$\dfrac{\dfrac{A \qquad B}{A \land B}\,[\land_I] \qquad\qquad A}{(A \land B) \land A}\,[\land_I]$$

Example (2)

Sequent:

$$\vdash (A \wedge B) \Rightarrow (B \wedge A)$$

Proof tree:

$$\dfrac{\dfrac{\dfrac{\cancel{A \wedge B}^{1}}{B}\,[\wedge_{E2}] \qquad \dfrac{\cancel{A \wedge B}^{1}}{A}\,[\wedge_{E1}]}{B \wedge A}\,[\wedge_{I}]}{(A \wedge B) \Rightarrow (B \wedge A)}\,[\Rightarrow_{I}, 1]$$

Example (3)

Sequent:

$$\vdash A \Rightarrow (B \Rightarrow (A \land B))$$

Proof tree:

$$\dfrac{\dfrac{\dfrac{\cancel{A}^2 \qquad \cancel{B}^1}{A \land B}\,[\land_I]}{B \Rightarrow (A \land B)}\,[\Rightarrow_I, 1]}{A \Rightarrow (B \Rightarrow (A \land B))}\,[\Rightarrow_I, 2]$$

Notice that both A and B were discharged/canceled by two instances of the application introduction rule ($\Rightarrow_I$).

Classical Logic vs. Intuitionistic Logic

In CL, a declarative statement is either true or false, whether or not we (or anybody else) know it, prove it, or verify it in any possible way.

In CL, *false* means the same as *not true*. This is expressed by the excluded middle principle: $A \vee \neg A$ holds no matter the meaning of A.

But the information content in $A \vee \neg A$ is limited. Consider the following:

$A \equiv$ *There are seven 7's in a row in the decimal representation of* π.

Maybe nobody will ever be able to determine the truth or falsity of A above. Yet, in CL, we are forced to accept that $A \vee \neg A$ must necessarily hold.

Classical Logic vs. Intuitionistic Logic

In many applications we want to find an actual solution to a problem, and not only to know that some solution exists.

We thus need to work the proof methods that provide actual solutions

The logic that accepts only "constructive” reasoning is intuitionistic logic.

The philosophical foundations of intuitionistic logic can be expressed in a very simplified way by

There is no absolute truth, there is only the knowledge and intuitive construction of the idealized mathematician, the creative subject. A logical judgement is only considered true if the creative subject can verify its correctness. Accepting this point of view inevitably leads to the rejection of the excluded middle as a uniform principle (Lectures on the Curry-Howard Isomorphism, M. H. Sorensen and P. Urzyczyn.)

Syntax of IPL

The language of *intuitionistic propositional logic* (IPL) is the same as the classical propositional logic.

We assume an infinite set $\mathcal{P}$ of *propositional letters* and we define the set Prop of formulas as the least set such that:

- if $\mathsf{A} \in \mathcal{P}$ then $\mathsf{A} \in \textsc{Prop}$
- if $\bot, \top \in \textsc{Prop}$
- if $\varphi, \psi \in \textsc{Prop}$ then $(\varphi \Rightarrow \psi)$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi) \in \textsc{Prop}$

The connectives are $\Rightarrow$ (implication), $\vee$ (disjunction), and $\wedge$ (conjunction).

$\neg\varphi$ abbreviates $\varphi \Rightarrow \bot$

$(\varphi \Leftrightarrow \psi)$ abbreviates $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Semantics of IPL (informal)

Intuitionistic logic does not follow the classical notion of "truth"

A judgement about a logical statement should be based on our ability to justify it via an explicit proof or "construction” and not by means of truth-tables (as it is normally done for classical propositional logic).

We should explain the meaning of compound formulas in terms of their constructions.

Such an explanation is often given by means of the informal interpretation known as the **BHK** (for Brouwer- Heyting-Kolmogorov)

Semantics (informal)

The BHK interpretation can be given by the following set of rules

- A construction of $\varphi_1 \wedge \varphi_2$ consists of a construction of φ_1 and a construction of φ_2;
- A construction of $\varphi_1 \vee \varphi_2$ consists of an indicator $\mathsf{i} \in \{1, 2\}$ and a construction of φ_{i};
- A construction of $\varphi_1 \Rightarrow \varphi_2$ is a method (function) transforming every construction of φ_1 into a construction of φ_2;
- There is no construction of $\perp$.

We can ask about the construction of a propositional letter when it is replaced by a concrete statement

BHK interpretation does not provide a precise and complete description of constructive semantics, as the very notion of "construction" is informal.

The algorithmic aspect of BHK interpretation will lead us to the **Curry-Howard Isomorphism.**

CL x IL

The distinctive feature of proof in Intuitionistic Logic when compared with that of Classical Logic is the absence of the principle of proof by contradiction (captured by the RAA rule in ND)

$$\dfrac{\boxed{\begin{array}{c}\cancel{\neg\varphi}\\ \vdots\\ \bot\end{array}}}{\varphi}\qquad (\text{RAA})$$

In IL, if we have a construction taking from $\neg\varphi$ to absurdity ($\bot$), that does not gives us a construction of φ

$\neg\neg\varphi \Rightarrow \varphi$ **cannot** be proved in IPL

(In Classical logic we need RAA to prove the excluded middle $\varphi \vee \neg\varphi$.)

Examples

All the formulas below can be proved in ILP (they are also tautologies in CL)

1. $\bot \Rightarrow A$
2. $A \Rightarrow B \Rightarrow A$
3. $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$
4. $A \Rightarrow \neg\neg A$
5. $\neg\neg\neg A \Rightarrow \neg A$
6. $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$
7. $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$
8. $((A \wedge B) \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))$
9. $\neg\neg(A \vee \neg A)$
10. $(A \vee \neg A) \Rightarrow \neg\neg A \Rightarrow A$

Examples

The formulas below **do not have a BHK interpretation** (they are tautologies in CL and some are similar/ "dual"to formulas of the previous example that do have BHK interpretation)

1. $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$
2. $A \vee \neg A$
3. $\neg\neg A \Rightarrow A$
4. $(\neg A \Rightarrow \neg B) \Rightarrow (A \Rightarrow B)$
5. $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
6. $(A \Rightarrow B) \Rightarrow (\neg A \Rightarrow B) \Rightarrow B$
7. $((A \leftrightarrow B) \leftrightarrow C) \leftrightarrow (A \leftrightarrow (B \leftrightarrow C))$
8. $(A \Rightarrow B) \leftrightarrow (\neg A \vee B)$
9. $(A \vee B \Rightarrow A) \vee (A \vee B \Rightarrow B)$
10. $(\neg\neg A \Rightarrow A) \Rightarrow A \vee \neg A$

Formula (3) **seems** to express the same principle as $A \Rightarrow \neg\neg A$

Formula (4) and $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$ are often treated as two equivalent patterns of a proof by contradiction

Formula (5) and $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$ are known as *De Morgan's laws*, and express the classical duality between conjunction and disjunction.

Symmetry of CL disappears in BHK since A and $\neg\neg A$ are not identified with each other (as they are in CL)

Proofs in ND for ILP - sequent style

Sequent $\Gamma \vdash \varphi$:

- read as Γ proofs φ, or φ can be proved from Γ
- Γ is a finite set of formulas

Notation

- $\varphi, \psi \ \equiv \ \{\varphi, \psi\}$
- $\Gamma, \varphi \ \equiv \Gamma \cup \{\varphi\}$

Nodes of a proof tree for $\Gamma \vdash \varphi$ in this *sequent* style are *sequents*

- root: $\Gamma \vdash \varphi$
- leaves: axioms of the form $\Gamma', \psi \vdash \psi$

Rules of ND for IPL - sequent style

We adopt a style of rules where the nodes of the derivation tree are **sequents**. Each node of the derivation tree then carries with it the set of hypotheses that can still be considered in that point of the proof.

$$\Gamma, \varphi \vdash \varphi \quad \text{(Ax)}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \quad (\bot\text{-E})$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \quad (\wedge\text{-E}1)$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \quad (\vee\text{-I}1)$$

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \quad (\wedge\text{-I})$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \psi \vee \varphi} \quad (\vee\text{-I}2)$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \quad (\wedge\text{-E}2)$$

$$\frac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma \vdash \varphi \Rightarrow \xi \qquad \Gamma \vdash \psi \Rightarrow \xi}{\Gamma \vdash \xi} \quad (\vee\text{-E})$$

$$\frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \quad (\Rightarrow\text{-E})$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \quad (\Rightarrow\text{-I})$$

Proofs in ND for ILP

Proof that $\vdash \bot \Rightarrow A$ in the traditional ND style:

$$\cfrac{\cfrac{\cancel{\bot}}{A}\,\bot_E}{\bot \Rightarrow A}\,\Rightarrow_I$$

Proof that $\vdash \bot \Rightarrow A$ in the *sequent* ND style:

$$\cfrac{\cfrac{\cfrac{}{\bot \vdash \bot}\,Ax}{\bot \vdash A}\,\bot_E}{\varnothing \vdash \bot \Rightarrow A}\,\Rightarrow_I$$

Proofs in ND for ILP

Proof that $(A \wedge B) \Rightarrow (B \wedge A)$ in the traditional ND style:

$$\dfrac{\dfrac{\dfrac{\cancel{A \wedge B}}{B}\,[\wedge_{E2}] \qquad \dfrac{\cancel{A \wedge B}}{A}\,[\wedge_{E1}]}{B \wedge A}\,[\wedge_{I}]}{(A \wedge B) \Rightarrow (B \wedge A)}\,[\Rightarrow_{I}]$$

Proof that $(A \wedge B) \Rightarrow (B \wedge A)$ in the *sequent* ND style:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\mathrm{Ax}}{A \wedge B \vdash B}\,[\wedge_{E2}] \qquad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\mathrm{Ax}}{A \wedge B \vdash A}\,[\wedge_{E1}]}{A \wedge B \vdash B \wedge A}\,[\wedge_{I}]}{\varnothing \vdash (A \wedge B) \Rightarrow (B \wedge A)}\,[\Rightarrow_{I}]$$

Content

Isomorphism of Curry-Howard

In intuitionistic logic, it is possible to view proofs as programs and proposition as types

- a proof of formula $A \wedge B$ is a proof of A **paired** with a proof of B,
- a proof of formula $A \Rightarrow B$ is a **procedure/function** which transforms evidence for A into evidence for B
- Infering B from $A \Rightarrow B$ and A corresponds to what happens when we apply a function of type $A \Rightarrow B$ to an argument of type A.
- a proof of $A \vee B$ is a proof of A or a proof of B, and this proof is tagged so that we know whis is the case (like a union type in programming languages)
- What about the *ex falso quiodlibet* rule ?? it express that (we expect) that there is no proof of falsity (formula $\perp$ is like the empty data type)

Example

Consider the following proof of proposition A (with a *detour*). Assume that we have proof terms M_1 and M_2, for propositions A and B, respectively

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{M}_1 : \mathsf{A} \qquad \Gamma \vdash \mathsf{M}_2 : \mathsf{B}}{\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2 \rangle : \mathsf{A} \wedge \mathsf{B}}\,[\wedge_{\mathsf{I}}]}{\Gamma \vdash \mathsf{fst}\ \langle \mathsf{M}_1, \mathsf{M}_2 \rangle : \mathsf{A}}\,[\wedge_{\mathsf{E1}}]$$

The term $\mathsf{fst}\ \langle \mathsf{M}_1, \mathsf{M}_2 \rangle$ represents the proof above.

The example show only the static aspect of the isomorphism. What is the counterpart of beta-reduction in logic ?

It can be simplified, according to the reduction rules of the calculus:

$$\mathsf{fst}\ \langle \mathsf{M}_1, \mathsf{M}_2 \rangle \quad \rightarrow_\beta \quad \mathsf{M}_1$$

Proof normalisation means to simplify proofs, eliminating proof *detours*.

Isomorphism of Curry-Howard

Many proof related concepts can be interpreted in terms of computations and vice-versa.

Logic	**Lambda Calculus**
formula	type
proof	term
connective	**type constructor**
implication	function type
conjunction	product type
disjunction	disjoint sum
absurdity	empty type
introduction	constructor
elimination	destructor
assumption	term variable
propositional variable	type variable
provability	type inhabitation
normal proof	normal form
proof detour	redex
normalization	reduction

Lecture Notes on the Curry-Howard Isomorphism (chapter 4). Morten H. Sørensen and Pawel Urzyczyn

Isomorphism of Curry-Howard

The Curry-Howard isomorphism, also known as the “propositions-as- types” paradigm, establishes a correspondence between logic and computation.

It started with the observation that a logical formula $A \Rightarrow B$ corresponds to a type of functions from A to B.

Proving an implication $A \Rightarrow B$ after inferring B from hypothesis A is similar to constructing a function mapping any term of type A to term of type B

The **basic ideia** is that proofs in the propositional intuitionistic logic are represented by simply typed λ-terms.

Propositions as types

The following maps types of $\lambda_{\rightarrow}$ with extensions to formulas of IPL (recall that 0 is the empty type, and that 1 is the unit type):

$$\begin{array}{lcl}
\mathtt{TyToProp}(A) & = & A \\
\mathtt{TyToProp}(0) & = & \bot \\
\mathtt{TyToProp}(1) & = & \top \\
\mathtt{TyToProp}(\tau_1 \rightarrow \tau_2) & = & \mathtt{TyToProp}(\tau_1) \Rightarrow \mathtt{TyToProp}(\tau_2) \\
\mathtt{TyToProp}(\tau_1 \times \tau_2) & = & \mathtt{TyToProp}(\tau_1) \wedge \mathtt{TyToProp}(\tau_2) \\
\mathtt{TyToProp}(\tau_1 + \tau_2) & = & \mathtt{TyToProp}(\tau_1) \vee \mathtt{TyToProp}(\tau_2)
\end{array}$$

A typing context is mapped to a set of hypothesis as follows:

$$\mathsf{TyCxToHyp}([x_1 : \tau_1, ..., x_n : \tau_n]) \quad = \quad \mathtt{TyToProp}(\tau_1), ..., \mathtt{TyToProp}(\tau_n)$$

Isomorphism of Curry-Howard

The following captures the static part of the Isomorphism (for $\lambda_{\rightarrow}$ with extensions)

Proposition. (Curry Howard Isomorphism)

1. If $\Gamma \vdash \mathsf{M} : \tau$ in $\lambda_{\rightarrow}$ then $\Delta \vdash \varphi$ in IPL where $\Delta = \mathtt{TyCxtToHyp}(\Gamma)$ and $\varphi = \mathtt{TyToProp}(\tau)$

2. If $\Delta \vdash \varphi$ in IPL then there is term M and typing context Γ such that $\Gamma \vdash \mathsf{M} : \tau$ in $\lambda_{\rightarrow}$, where Γ and τ are such that $\mathtt{TyCxtToHyp}(\Gamma) = \Delta$, and $\mathtt{TyToProp}(\tau) = \Delta$.

Type Inhabitation problem for $\lambda_{\rightarrow}$: Is there a term M and typing context Γ such that $\Gamma \vdash \mathsf{M} : \tau$?

Inhabitation problem for type $\tau \equiv$ **provability problem** for the corresponding formula φ in IPL

Type Inhabitation

The correspondence between logic and lambda-calculus makes it possible to translate results concerning one of these systems into results about the other.

For instance, we have that :

- There is no combinator of type $((A \to B) \to A) \to A$ because Peirce's Law $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ is not valid in intuitionistic logic

- Formula $((((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B) \Rightarrow B$ is a theorem in intuitionistic logic because we can construct the following term in $\lambda_{\to}$ of type $((((A \to B) \to A) \to A) \to B) \to B$:

$$
\begin{array}{l}
\lambda x : (((A \to B) \to A) \to A) \to B. \\
\quad x\ [\ \lambda y : (A \to B) \to A. \\
\qquad\quad y\ [\ \lambda z : A. \\
\qquad\qquad\quad x\ [\ \lambda u : (A \to B) \to A.\ z] \\
\qquad\quad] \\
\quad]
\end{array}
$$

Isomorphism of Curry-Howard

Below we have both:

- Typing rules for function and for function application in $\lambda_{\rightarrow}$, and
- Natural deduction rules $\Rightarrow_{\mathsf{I}}$ and $\Rightarrow_{\mathsf{E}}$ in IPL with explicit proof construction

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x : \varphi .\, M : \varphi \Rightarrow \psi} \quad (\Rightarrow_{\mathsf{I}})$$

$$\frac{\Gamma \vdash M : \varphi \Rightarrow \psi \qquad \Gamma \vdash N : \varphi}{\Gamma \vdash M\, N : \psi} \quad (\Rightarrow_{\mathsf{E}})$$

Isomorphism of Curry-Howard - rules

$$\begin{array}{lcl} M & ::= & x \mid M\,N \mid \lambda x\!:\!\tau.\,M \\ & \mid & \langle M_1, M_2\rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \\ & \mid & \mathsf{left}\ M \mid \mathsf{right}\ M \mid \mathsf{case}\ M\,[x \mapsto M_1, y \mapsto M_2] \\ & \mid & \mathsf{bot}\ M \mid \langle\rangle \end{array}$$

$$\begin{array}{cccccccccccccc} \tau & ::= & A & \mid & \tau_1 \to \tau_2 & \mid & \tau_1 \times \tau_2 & \mid & \tau_1 + \tau_2 & \mid & 0 & \mid & 1 \\ \varphi & ::= & A & \mid & \varphi_1 \Rightarrow \varphi_2 & \mid & \varphi_1 \wedge \varphi_2 & \mid & \varphi_1 \vee \varphi_2 & \mid & \bot & \mid & \top \end{array}$$

$$\frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{fst}\ M : \varphi}\ (\wedge\text{-E1}) \qquad \frac{\Gamma \vdash M : \varphi \wedge \psi}{\Gamma \vdash \mathsf{snd}\ M : \psi}\ (\wedge\text{-E2}) \qquad \frac{\Gamma \vdash M_1 : \varphi \quad \Gamma \vdash M_2 : \psi}{\Gamma \vdash \langle M_1, M_2\rangle : \varphi \wedge \psi}$$

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \mathsf{left}\ M : \varphi \vee \psi}\ (\vee\text{-I1}) \qquad \frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \mathsf{right}\ M : \psi \vee \varphi}\ (\vee\text{-I2})$$

$$\frac{\Gamma \vdash M : \varphi \vee \psi \qquad \Gamma, x : \varphi \vdash M_1 : \xi \qquad \Gamma, y : \psi \vdash M_2 : \xi}{\Gamma \vdash \mathsf{case}\ M\,[x \mapsto M_1, y \mapsto M_2] : \xi}\ (\vee\text{-E})$$

$$\frac{\Gamma \vdash M : \varphi \Rightarrow \psi \qquad \Gamma \vdash N : \varphi}{\Gamma \vdash M\,N : \psi}\ (\Rightarrow\text{-E}) \qquad \frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x : \varphi.\ M :\ \varphi \Rightarrow \psi}\ (\Rightarrow\text{-I})$$

Isomorphism of Curry-Howard - rules (cont.)

$$\begin{array}{lcl} \mathsf{M} & ::= & \ldots \mid \mathsf{bot}\ \mathsf{M} \mid \langle\rangle \\ \tau & ::= & \ldots \mid 0 \mid 1 \\ \varphi & ::= & \ldots \mid \bot \mid \top \end{array}$$

$$\Gamma, x : \varphi \vdash x : \varphi \quad (\mathrm{Ax}) \qquad \Gamma \vdash \langle\rangle : \top \quad (\mathrm{AxT}) \qquad \frac{\Gamma \vdash \mathsf{M} : \bot}{\Gamma \vdash \mathsf{bot}\ \mathsf{M} : \varphi} \quad (\bot\text{-E})$$

- **unit** type 1 has term $\langle\rangle$ as single inhabitant. Term $\langle\rangle$ is the constructor for type 1
  - as expected, term $\langle\rangle$ is the **proof term** for proposition $\top$

- **empty** type 0 has no inhabitant. Type 0 has no constructor
  - but it has been given a destructor, bot, so we can represent a proof term that would follow if we construct a proof leading to absurdity

Example

Proof term for proposition $(A \land B) \Rightarrow (B \land A)$;

$$\dfrac{\dfrac{\dfrac{\dfrac{}{z\colon A \land B \vdash z : A \land B}[\mathsf{Ax}]}{z\colon A \land B \vdash \mathsf{snd}\ z : B}[\land_{E2}] \qquad \dfrac{\dfrac{}{z\colon A \land B \vdash z : A \land B}[\mathsf{Ax}]}{A \land B \vdash \mathsf{fst}\ z : A}[\land_{E1}]}{z\colon A \land B \vdash \langle \mathsf{snd}\ z, \mathsf{fst}\ z \rangle : B \land A}[\land_{I}]}{\varnothing \vdash \lambda z : A \times B.\ \langle \mathsf{snd}\ z, \mathsf{fst}\ z \rangle : (A \land B) \Rightarrow (B \land A)}[\Rightarrow_{I}]$$

Content

Polymorphic Lambda Calculus

Polymorphism

Polymorphism means “many forms”. Here we will discuss this notion in the context of type systems, i.e. a single term that can be used at many types (via instantiation).

In the STLC $\lambda_{\rightarrow}$, we have terms and types, however abstraction and application only occur at the term level.

- An expression $(\lambda x{:}\tau.M)$ represents a **term** M that *depends* on other term (referred as x) to be “complete”.
- By constructing the application $(\lambda x{:}\tau.M)\ N$, one passes N as an argument for $(\lambda x{:}\tau.M)$, “completing it” and obtaining the contractum $[N/x]M$ by beta-reduction.

We can say that the notion of **terms that depend on terms** is represented by the presence of a function constructor (term abstraction $\lambda x{:}\tau.M$) and a function destructor (term application $M\ N$).

Motivating example: identity

Consider the identity term in the untyped lambda calculus: $\lambda x.x$. Notice that it is a unique term, applicable to all other terms in the untyped lambda calculus λ.

Consider now the identity term(s) in the simply typed lambda calculus $\lambda_{\rightarrow}$:

- In the Church style, we have one identity for **each** type τ:

$$\begin{array}{lll} \lambda x{:}A.\ x & \lambda x{:}B.\ x & \lambda x{:}B \rightarrow A.\ x \\ \lambda x{:}A \rightarrow A.\ x & \lambda x{:}(B \rightarrow A) \rightarrow A.\ x & \ldots \end{array}$$

  There is not a single, unique identity function: they all differ due to a distinct type τ in their annotations of x. They all, however, have type $\tau \rightarrow \tau$.

- In the Curry style, there is a unique identity function $\lambda x.x$. Although unique, notice that it has an infinite number of types: $A \rightarrow A$, $B \rightarrow B$, $(B \rightarrow A) \rightarrow (B \rightarrow A)$, …

Motivating example: identity (2)

Either having infinite terms (Church style) or a single term with infinite types (Curry style), there is a pattern to the type of the identity: it is always $\tau \to \tau$ for some type τ.

This common type pattern can be captured by introducing **terms that depend on types**. Starting with the STLC, we introduce two new syntatic forms: a type abstraction $\Lambda X.M$ and a type application $M\langle\tau\rangle$, resulting in the **polymorphic lambda calculus** $\lambda 2$.

In $\lambda 2$ we can describe a unique term that represent all possible identities of the Curry and Church style, and with a single type:

$$(\Lambda X.\ \lambda x : X.\ x) \quad : \quad \forall X.X \to X$$

Type abstraction and type instantiation are the essence of **parametric polymorphism** (also known as *generic types*), feature that occurs in many modern functional programming languages such as Haskell, Ocaml, F#, Scala, among others.

History

The calculus λ_2 was proposed independently by two authors:

- Jean Yves Girard (1972), from a logical perspective, named **System F**
- John Reynolds (1974), from a computational perspective, named **polymorphic lambda calculus**

J. Y. Girard

Important results are attributed to them:

- the abstraction theorem by Reynolds
- the representation theorem and strong normalization by Girard

J. Reynolds

This connection between logic and computation can be seen as another instance of the Curry-Howard isomorphism (or, in this case, Girard-Reynolds isomorphism).

Syntax of (pre-)types

We assume a pre-existing infinite set of **type variables** $\mathsf{TypeVar}$, whose elements we denote by uppercase letters $\mathsf{X}, \mathsf{Y}, \mathsf{Z} \ldots$.

Notice: uppercase letters $\mathsf{X}, \mathsf{Y}, \mathsf{Z} \ldots$ will always denote **type variables**, not *atomic types* (which will not occur in this version of $\lambda 2$) or *terms* (which will receive another meta-variable).

Definition: A **polymorphic pre-type** is an element of the set $\mathbb{T}_2^-$, constructed by the following syntax:

$$\begin{array}{lll} \tau & \in & \mathbb{T}_2^- \\ \tau & ::= & \mathsf{X} \mid \tau_1 \to \tau_2 \mid \forall \mathsf{X}.\tau_1 \end{array}$$

We use the meta-variable τ to range over polymorphic pre-types (and, later, also for types).

Syntax of (pre-)terms

From $\lambda 2$ onwards, we will only consider Church-style lambda terms (with explicit type annotations).

Definition: The set Λ_2^- of pre-terms is defined by the following abstract grammar:

$$\begin{array}{lll} e & \in & \Lambda_2^- \\ e & ::= & x \mid e_1\, e_2 \mid \lambda x : \tau . e_1 \\ & & \Lambda X . e_1 \qquad\qquad\qquad \text{(type abstraction)} \\ & & e_1 \langle \tau \rangle \qquad\qquad\qquad\quad \text{(type application)} \end{array}$$

Notice:

- we now use e for (pre-)terms, to avoid confusion with type variables.
- type abstraction $\Lambda X.e_1$ represents a term depending on a (pre-)type X.
- type application $e_1\langle\tau\rangle$ represents passing (pre-)type τ as argument for (pre-)term e_1.

Free variables

Since there are both **(term) variables** $x, y, z, \ldots$ and **type variables** $X, Y, Z, \ldots$ in $\lambda 2$ terms, we need a few more functions to collect both *free variables* and *free type variables*.

Free variables of a pre-term:

$$\begin{array}{lcl} FV & : & \Lambda_2^- \to \mathcal{P}(Var) \\ FV(x) & = & \{x\} \\ FV(\lambda x : \tau . e_1) & = & FV(e_1) - \{x\} \\ FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(\Lambda X . e_1) & = & FV(e_1) \\ FV(e_1 \langle \tau \rangle) & = & FV(e_1) \end{array}$$

Free type variables of a pre-type:

$$\begin{array}{lcl} FTV & : & \mathbb{T}_2^- \to \mathcal{P}(TypeVar) \\ FTV(X) & = & \{X\} \\ FTV(\tau_1 \to \tau_2) & = & FTV(\tau_1) \cup FTV(\tau_2) \\ FTV(\forall X . \tau_1) & = & FTV(\tau_1) - \{X\} \end{array}$$

Free variables (2)

Free type variables of a pre-term:

$$\begin{array}{lcl}
\mathsf{FTV} & : & \Lambda_2^- \to \mathcal{P}(\mathsf{TypeVar}) \\
\mathsf{FTV}(x) & = & \{\} \\
\mathsf{FTV}(\lambda x{:}\tau.e_1) & = & \mathsf{FTV}(\tau) \cup \mathsf{FTV}(e_1) \\
\mathsf{FTV}(e_1\ e_2) & = & \mathsf{FTV}(e_1) \cup \mathsf{FTV}(e_2) \\
\mathsf{FTV}(\Lambda X.e_1) & = & \mathsf{FTV}(e_1) - \{X\} \\
\mathsf{FTV}(e_1\langle\tau\rangle) & = & \mathsf{FTV}(e_1) \cup \mathsf{FTV}(\tau)
\end{array}$$

The α-equivalence relation on both pre-types and pre-terms is defined as usual, allowing for the renaming of

- bound type variables (in pre-types)
- bound variables and bound type variables (in pre-terms)

We define the **set of terms** Λ_2 and the **set of types** $\mathbb{T}_2$ as, respectively, the sets of equivalence classes of α-equivalent pre-terms and pre-types. We remark that α-equivalence is compatible with the FV, FTV and substitution operations (to be defined).

Substitution

We have three kinds of substitution in the polymorphic lambda calculus:

- substitution of a variable by a term in a term
- substitution of a type variable by a type in a type
- substitution of a type variable by a type in a term

We reuse the same notation $[\_/\_]\_$ for the three substitutions above, which can be differentiated through their arguments.

The next definitions **assume** the *Barendregt Assumption*, i.e. that all bound variables and bound type variables in **terms** and **types** have fresh, unique names that do not occur anywhere else (including other terms or environments). This allows us to reason about substitutions without worrying about the capture of free variables.

Notice: in concrete implementations the *Barendregt Assumption* must be *enforced*, i.e. we should always perform α-renaming when we detect a possible name conflict (as shown in the substitution of pre-terms in λ).

Substitution (2)

Substitution of a type variable by a type in a type:

$$\begin{array}{lcl}
[\_/\_]\_ & : & \mathbb{T}_2 \times \mathsf{TypeVar} \times \mathbb{T}_2 \to \mathbb{T}_2 \\
[\tau/Y]X & = & \begin{cases} \tau & \text{if } X = Y \\ X & \text{if } X \neq Y \end{cases} \\
[\tau/Y](\tau_1 \to \tau_2) & = & [\tau/Y]\tau_1 \to [\tau/Y]\tau_2 \\
[\tau/Y](\forall X.\tau_1) & = & \forall X.[\tau/Y]\tau_1
\end{array}$$

Substitution (3)

Substitution of a type variable by a type in a term:

$$
\begin{array}{lcl}
[\_/\_]\_ & : & \mathbb{T}_2 \times \mathsf{TypeVar} \times \Lambda_2 \to \Lambda_2 \\
[\tau/Y](x) & = & x \\
[\tau/Y](\lambda x : \tau'.e_1) & = & \lambda x : [\tau/Y]\tau' . [\tau/Y]e_1 \\
[\tau/Y](e_1\ e_2) & = & [\tau/Y]e_1\ [\tau/Y]e_2 \\
[\tau/Y](\Lambda X.e_1) & = & \Lambda X.[\tau/Y]e_1 \\
[\tau/Y](e_1\langle\tau_1\rangle) & = & [\tau/Y]e_1\ \langle[\tau/Y]\tau_1\rangle
\end{array}
$$

Substitution (4)

Substitution of a variable by a term in a term:

$$
\begin{array}{lcl}
[\_/\_]\_ & : & \Lambda_2 \times \mathsf{Var} \times \Lambda_2 \to \Lambda_2 \\
[e/y](x) & = & \begin{cases} e & \text{if } x = y \\ x & \text{otherwise} \end{cases} \\
[e/y](\lambda x : \tau . e_1) & = & \lambda x : \tau . [e/y] e_1 \\
[e/y](e_1\ e_2) & = & [e/y]e_1\ [e/y]e_2 \\
[e/y](\Lambda X . e_1) & = & \Lambda X . [e/y] e_1 \\
[e/y](e_1 \langle \tau_1 \rangle) & = & ([e/y]e_1)\ \langle \tau_1 \rangle
\end{array}
$$

Type environments

Type environments are more complex in λ_2 than in $\lambda_\to$ because they may declare both *variables* and *type variables.*

The *order of declarations* will matter, and therefore we will treat them as *lists.*

Let Env be the set of environments defined by the following grammar:

$$\begin{array}{lcl} \Gamma & \in & \mathsf{Env} \\ \Gamma & ::= & \bullet \mid \Gamma', x:\tau \mid \Gamma', X:\star \end{array}$$

Notation: we usually omit the $\bullet$ when writing non-empty environments, i.e. we write $X:\star, x:X$ instead of $\bullet, X:\star, x:X$.

Example: $X:\star, x:X \to X, Y:\star, z:X \to Y$

Contexts

Definition: **domain** of an environment.

$$\begin{array}{lcl} \mathsf{dom} & : & \mathsf{Env} \to \mathcal{P}(\mathsf{Var} \cup \mathsf{TypeVar}) \\ \mathsf{dom}(\bullet) & = & \{\} \\ \mathsf{dom}(\Gamma, \mathsf{X} : \star) & = & \mathsf{dom}(\Gamma) \cup \{\mathsf{X}\} \\ \mathsf{dom}(\Gamma, \mathsf{x} : \tau) & = & \mathsf{dom}(\Gamma) \cup \{\mathsf{x}\} \end{array}$$

A type environment may be well-formed or ill-formed. The well-formed environments are constructed as follows:

- the statement $\mathsf{X} : \star$ declares that the type variable X is available;
- the statement $\mathsf{x} : \tau$ declares that the variable x is of type τ;
- when introducing a new statement $\mathsf{x} : \tau$, all type variables occurring in τ must have been previouly declared in the type environment.

Judgements

We will have three kinds of judgements in λ_2:

$\Gamma \vdash \mathbf{ok}$	denotes that Γ is well-formed (a context)
$\Gamma \vdash \tau : \star$	denotes that τ is well-formed under the context Γ
$\Gamma \vdash e : \tau$	denotes that e is of type τ under context Γ

Their signature (as relations) are, respectively,

$$(\_ \vdash \mathbf{ok}) \subseteq \mathsf{Env}$$

$$(\_ \vdash \_ : \star) \subseteq \mathsf{Env} \times \mathbb{T}_2$$

$$(\_ \vdash \_ : \_) \subseteq \mathsf{Env} \times \Lambda_2 \times \mathbb{T}_2$$

Context and type formation

Context formation

$$\frac{}{\bullet \vdash \mathbf{ok}} \quad (\textsc{emptyOK})$$

$$\frac{\Gamma \vdash \mathbf{ok} \qquad X \notin \mathrm{dom}(\Gamma)}{\Gamma, X{:}\star \vdash \mathbf{ok}} \quad (\textsc{tvarOK})$$

$$\frac{\Gamma \vdash \tau : \star \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x{:}\tau \vdash \mathbf{ok}} \quad (\textsc{varOK})$$

Notice: in varOK, the premisse $\Gamma \vdash \tau : \star$ requires Γ to be well-formed.

Type formation

$$\frac{\Gamma, X{:}\star, \Gamma' \vdash \mathbf{ok}}{\Gamma, X{:}\star, \Gamma' \vdash X : \star} \quad (\textsc{tvarForm})$$

$$\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \to \tau_2 : \star} \quad (\textsc{arrowForm})$$

$$\frac{\Gamma, X{:}\star \vdash \tau : \star}{\Gamma \vdash \forall X.\tau : \star} \quad (\textsc{forallForm})$$

Notice: intuitively, $\Gamma \vdash \tau : \star$ means that all free type variables occurring in τ are declared in Γ.

Type system

$$\frac{\Gamma, x : \tau, \Gamma' \vdash \mathbf{ok}}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \quad (\textsc{var})$$

$$\frac{\Gamma \vdash e : \forall X.\tau' \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash e\ \langle\tau\rangle : [\tau/X]\tau'} \quad (\textsc{typeApp})$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.e) : \tau \to \tau'} \quad (\textsc{abs})$$

$$\frac{\Gamma, X : \star \vdash e : \tau}{\Gamma \vdash \Lambda X.e : \forall X.\tau} \quad (\textsc{typeAbs})$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'} \quad (\textsc{app})$$

Type derivations, legal terms and types

Example: of a type derivation in λ_2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{X{:}\star, Y{:}\star, y{:}Y \vdash \mathbf{ok}}}{X{:}\star, Y{:}\star, y{:}Y \vdash y : Y}\,(\textsc{var})}{X{:}\star, Y{:}\star \vdash (\lambda y{:}Y.y) : Y \to Y}\,(\textsc{abs})}{X{:}\star \vdash (\Lambda Y.\lambda y{:}Y.y) : \forall Y.Y \to Y}\,(\textsc{typeAbs}) \qquad \dfrac{\dfrac{\vdots}{X{:}\star \vdash \mathbf{ok}}}{X{:}\star \vdash X : \star}\,(\textsc{tvarForm})}{X{:}\star \vdash (\Lambda Y.\lambda y{:}Y.y)\,\langle X\rangle : X \to X \quad = [X/Y](Y \to Y)}\,(\textsc{typeApp})$$

We omitted the derivations of well-formed contexts for reasons of space.

We say a type τ is **well-formed** (or legal) in λ_2 if there is a context Γ such that $\Gamma \vdash \tau : \star$ (according to the type formation rules).

We say a term e is **well-typed** (or legal) in λ_2 if there are Γ and $\tau \in \mathbb{T}_2$ such that $\Gamma \vdash e : \tau$.

Type erasure

How do terms of the polymorphic lambda calculus λ_2 relate to terms of the untyped lambda calculus λ?

We can define a **type erasure** function $|\_| : \Lambda_2 \to \Lambda$ that simply removes annotations, type abstractions and instantiations.

$$\begin{array}{lcl} |\_| & : & \Lambda_2 \to \Lambda \\ |x| & = & x \\ |\lambda x\!:\!\tau.e_1| & = & \lambda x.|e_1| \\ |e_1\ e_2| & = & |e_1|\ |e_2| \\ |\Lambda X.e_1| & = & |e_1| \\ |e_1\langle\tau\rangle| & = & |e_1| \end{array}$$

We say a untyped term $e \in \Lambda$ is **typable in** Λ_2 if there is a term $e' \in \Lambda_2$, such that $|e'| = e$ and e' is well-typed in λ_2.

Typable terms in λ_2

The self application term $(\lambda x.x\ x)$ is typable in λ_2 by choosing adequate type annotations and type instantiations:

- $\vdash (\lambda x : \forall C.C \to C.\ x\ \langle \forall C.C \to C \rangle\ x) : (\forall C.C \to C) \to (\forall C.C \to C)$

Exercise:

- write the derivation tree for the typing judgement above.
- find another well-typed λ_2-term e such that $|e| = \lambda x.x\ x$.

However, there are also terms in λ that are not typable in Λ_2. Examples include:

- $(\lambda x.x\ x)\ (\lambda x.x\ x)$
- $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$

Redexes

There are two kinds of redexes in Λ_2:

- $(\lambda x : \tau . e)\ e'$
- $(\Lambda X . e)\ \langle \tau \rangle$

whose contractums are, respectively,

- $[e'/x]e$
- $[\tau/X]e$

These redexes can occur anywhere in a larger term, and will be used to define the respective notion of beta-reduction $\to_\beta$ (as follows).

Semantics

Beta reduction $(\_ \to_\beta \_) \subseteq \Lambda_2 \times \Lambda_2$ is the smallest relation satisfying:

$$\frac{}{(\lambda x{:}\tau.e)\ e' \to_\beta [e'/x]e}\ (\beta) \qquad \frac{e_1 \to_\beta e_1'}{e_1\ e_2 \to_\beta e_1'\ e_2}\ (\textsc{cpApp1}) \qquad \frac{e \to_\beta e'}{\Lambda X.e \to_\beta \Lambda X.e'}\ (\textsc{cpTyLam})$$

$$\frac{}{(\Lambda X.e)\ \langle\tau\rangle \to_\beta [\tau/X]e}\ (\textsc{inst}) \qquad \frac{e_2 \to_\beta e_2'}{e_1\ e_2 \to_\beta e_1\ e_2'}\ (\textsc{cpApp2}) \qquad \frac{e \to_\beta e'}{e\ \langle\tau\rangle \to_\beta e'\ \langle\tau\rangle}\ (\textsc{cpTyApp})$$

$$\frac{e \to_\beta e'}{\lambda x{:}\tau.e \to_\beta \lambda x{:}\tau.e'}\ (\textsc{cpLam})$$

Notice:

- only terms are transformed by reductions. There is *substitution* of type variables by types, but not *reductions* on types.
- after defining $\to_\beta$, we define $\twoheadrightarrow_\beta$ and $=_\beta$ in the usual way.

Programming in λ_2

λ_2 is more expressive (with respect to typable terms) than $\lambda_{\rightarrow}$.

We will consider now how to represent some datatypes and functions over them using the well-typed terms of λ_2:

- booleans / conditionals
- naturals / natural iterators
- pairs / projections
- lists / list iterators
- empty and unit

Each datatype will have a set of constructors (to insert information into/build the data structure) and iterators (that extract information from the data structure).

Notice: certainly it is more convenient to introduce the datatypes above as primitive constructs (as in System T and PCF). Here we just intend to show that it is possible to represent them in the pure version of λ_2.

Booleans

$$
\begin{array}{lcl}
\mathsf{bool} & = & \forall C. C \to C \to C \\
\\
\mathbf{true} & = & \Lambda C.\ \lambda a{:}C.\ \lambda b{:}C.\ a \\
\mathbf{false} & = & \Lambda C.\ \lambda a{:}C.\ \lambda b{:}C.\ b
\end{array}
$$

Compare this encoding with the ones from $\lambda_{\to}$, in which each atomic type had a version of the booleans.

In Λ_2, each distinct *instantiation* $\mathsf{true}\ \langle\tau\rangle$ and $\mathsf{false}\ \langle\tau\rangle$ results in a particular true and false for the specific type τ, exactly as defined in $\lambda_{\to}$.

Example:

$$
\begin{array}{lcl}
\mathbf{true}\ \langle U\rangle & =_{\beta} & \lambda a{:}U.\ \lambda b{:}U.\ a \quad : \quad (U \to U \to U) \\
\mathbf{true}\ \langle A\rangle & =_{\beta} & \lambda a{:}A.\ \lambda b{:}A.\ a \quad : \quad (A \to A \to A)
\end{array}
$$

Conditional expressions

The `if-then-else` construct has three arguments:

- a boolean value b
- a *then*-expression e_1
- an *else*-expression e_2

with the remark that the *then*-expression and the *else*-expression should be of the same type.

$$\begin{aligned}
\textbf{if} \quad &: \quad \forall D.\ \overbrace{(\forall C.C \to C \to C)}^{\mathsf{bool}} \to D \to D \to D \\
\textbf{if} \quad &= \quad \Lambda D.\ \lambda b : \overbrace{(\forall C.C \to C \to C)}^{\mathsf{bool}}\ .\ \lambda e_1 : D.\ \lambda e_2 : D.\ b\ \langle D \rangle\ e_1\ e_2
\end{aligned}$$

Example: **if** $\langle \mathsf{nat} \rangle$ **true** 2 7

Exercise: define **and**, **or** and **not** in $\lambda 2$.

Natural numbers

$$\begin{array}{lcl}
\mathsf{nat} & = & \forall \mathsf{C}.(\mathsf{C} \to \mathsf{C}) \to \mathsf{C} \to \mathsf{C} \\
\\
\mathbf{0} & = & \Lambda \mathsf{C}.\ \lambda \mathsf{f} : \mathsf{C} \to \mathsf{C}.\ \lambda \mathsf{x} : \mathsf{C}.\ \mathsf{x} \\
\mathbf{1} & = & \Lambda \mathsf{C}.\ \lambda \mathsf{f} : \mathsf{C} \to \mathsf{C}.\ \lambda \mathsf{x} : \mathsf{C}.\ \mathsf{f}\ \mathsf{x} \\
\mathbf{2} & = & \Lambda \mathsf{C}.\ \lambda \mathsf{f} : \mathsf{C} \to \mathsf{C}.\ \lambda \mathsf{x} : \mathsf{C}.\ \mathsf{f}\ (\mathsf{f}\ \mathsf{x}) \\
 & \vdots & \\
\mathsf{succ} & = & \lambda \mathsf{n} : \mathsf{nat}.\ \Lambda \mathsf{C}.\ \lambda \mathsf{f} : \mathsf{C} \to \mathsf{C}.\ \lambda \mathsf{x} : \mathsf{C}.\ \mathsf{f}\ (\mathsf{n}\ \langle \mathsf{C} \rangle\ \mathsf{f}\ \mathsf{x})
\end{array}$$

Notice that the representation above can be seen as an annotation of the **Church encoding** of natural numbers, in which we (i) add an extra type abstraction at the beginning of the number, and (ii) require an instantiation when applying the number to arguments.

Exercise: define **add**, **mult** and **isZero** in $\lambda 2$

Iterating over natural numbers

Several functions on natural numbers can be defined by the following recursion scheme (in pseudocode):

```
foo (x:nat) : output_type =
case x of
  0       =>    base_result
  succ y  =>    expression_using_(foo y)
```

Example: we can define the function double : $\mathbb{N} \to \mathbb{N}$ in this style:

```
double (x:nat) : nat =
case x of
  0       =>  0
  succ y  =>  succ (succ (double y))
```

As with Church numerals, polymorphic naturals create a sequence of applications of a particular operation over a starting value. The number of applications reflect the encoded number.

Iterating over natural numbers (2)

By passing as arguments for a polymorphic natural number

- an output type O
- a base case (of type O)
- a step expression as a function (of type $\mathsf{O} \to \mathsf{O}$) that receives as argument the result of the recursive call over the predecessor

we can express recursive functions that *consume* natural numbers and *return* values of any other kind.

Example:

$$\begin{aligned} \mathbf{double} &= \lambda \mathsf{n}:\mathsf{nat}.\ \mathsf{n}\ \langle \mathsf{nat} \rangle\ (\lambda \mathsf{n}:\mathsf{nat}.\ \mathbf{succ}\ (\mathbf{succ}\ \mathsf{n}))\ \mathbf{0} \\ \mathbf{even} &= \lambda \mathsf{n}:\mathsf{nat}.\ \mathsf{n}\ \langle \mathsf{bool} \rangle\ \mathbf{not}\ \mathbf{true} \end{aligned}$$

Notice:

- the argument order is output_type, step and base.
- each polymorphic natural, when applied, works as an **iterator**.

Pairs

We start by fixing types A and B (for now, later we generalize them).

The **product type** $\mathsf{A} \times \mathsf{B}$ will be represented by the *polymorphic type*

$$\forall \mathsf{C}.(\mathsf{A} \to \mathsf{B} \to \mathsf{C}) \to \mathsf{C}$$

The **constructor** $\mathsf{pair}_{\mathsf{AB}} : \mathsf{A} \to \mathsf{B} \to (\mathsf{A} \times \mathsf{B})$ is encoded by the term

$$\begin{array}{lcl} \mathsf{pairAB} & : & \mathsf{A} \to \mathsf{B} \to (\forall \mathsf{C}.(\mathsf{A} \to \mathsf{B} \to \mathsf{C}) \to \mathsf{C}) \\ \mathsf{pairAB} & = & \lambda \mathsf{a}:\mathsf{A}.\ \lambda \mathsf{b}:\mathsf{B}.\ (\Lambda \mathsf{C}.\ \lambda \mathsf{f}:\mathsf{A} \to \mathsf{B} \to \mathsf{C}.\ \mathsf{f\ a\ b}) \end{array}$$

Each choice of types A and B has a particular constructor pairAB. For example:

$$\begin{array}{lcl} \mathsf{pairNatBool} & : & \mathsf{nat} \to \mathsf{bool} \to (\forall \mathsf{C}.(\mathsf{nat} \to \mathsf{bool} \to \mathsf{C}) \to \mathsf{C}) \\ \mathsf{pairNatBool} & = & \lambda \mathsf{a}:\mathsf{nat}.\ \lambda \mathsf{b}:\mathsf{bool}.\ (\Lambda \mathsf{C}.\ \lambda \mathsf{f}:\mathsf{nat} \to \mathsf{bool} \to \mathsf{C}.\ \mathsf{f\ a\ b}) \end{array}$$

Pairs (2)

As in λ, we must pass a selector to a pair $p : A \times B$, and this function will choose either the element $a : A$ or the element $b : B$ stored in p.

The main distinction of $\lambda 2$ with respect to λ is the need to care for the proper instantiation of the return type.

The **projections** $\mathsf{fst}_{AB} : (A \times B) \to A$ and $\mathsf{snd}_{AB} : (A \times B) \to B$ are encoded by

$$\begin{aligned} \mathsf{fstAB} &= \lambda p{:}(\forall C.(A \to B \to C) \to C).\ p\ \langle A \rangle\ (\lambda a{:}A.\ \lambda b{:}B.\ a) \\ \mathsf{sndAB} &= \lambda p{:}(\forall C.(A \to B \to C) \to C).\ p\ \langle B \rangle\ (\lambda a{:}A.\ \lambda b{:}B.\ b) \end{aligned}$$

Example:

$\mathsf{fstNatBool} = \lambda p{:}(\forall C.(\mathsf{nat} \to \mathsf{bool} \to C) \to C).\ p\ \langle \mathsf{nat} \rangle\ (\lambda a{:}\mathsf{nat}.\ \lambda b{:}\mathsf{bool}.\ a)$

$\mathsf{fstNatBool}\ (\mathsf{pairNatBool}\ 7\ \mathsf{false}) =_\beta 7$

Pairs (3)

Given that we have type abstraction, we can generalize the types A and B, allowing for a unique definition of pair, fst and snd.

$$\begin{aligned}
\mathsf{pair} &= \Lambda A.\ \Lambda B.\ \lambda a{:}A.\ \lambda b{:}B.\ (\Lambda C.\ \lambda f{:}A \to B \to C.\ f\ a\ b) \\
\mathsf{fst} &= \Lambda A.\ \Lambda B.\ \lambda p{:}(\forall C.(A \to B \to C) \to C).\ p\ \langle A\rangle\ (\lambda a{:}A.\ \lambda b{:}B.\ a) \\
\mathsf{snd} &= \Lambda A.\ \Lambda B.\ \lambda p{:}(\forall C.(A \to B \to C) \to C).\ p\ \langle B\rangle\ (\lambda a{:}A.\ \lambda b{:}B.\ b)
\end{aligned}$$

The only drawback of this approach is the need to always instantiate the operations with the types of their arguments.

Example: $\mathsf{fst}\ \langle \mathsf{nat}\rangle\ \langle \mathsf{bool}\rangle\ (\mathsf{pair}\ \langle \mathsf{nat}\rangle\ \langle \mathsf{bool}\rangle\ 7\ \mathsf{false}) =_\beta 7$

Predecessor

We can replicate the definition of the **predecessor** function (as shown in λ) with polymorphic types:

- $\mathsf{output} = (\mathsf{nat} \times \mathsf{nat})$ (encoded using polymorphic types)
- $\mathsf{base} = (\mathsf{pair}\ \langle\mathsf{nat}\rangle\ \langle\mathsf{nat}\rangle\ \mathbf{0}\ \mathbf{0})$
- $\mathsf{step} = \lambda \mathsf{p}{:}(\mathsf{nat} \times \mathsf{nat}).$
  $(\mathsf{pair}\ \langle\mathsf{nat}\rangle\ \langle\mathsf{nat}\rangle\ \ (\mathsf{snd}\ \langle\mathsf{nat}\rangle\ \langle\mathsf{nat}\rangle\ \mathsf{p})\ \ (\mathbf{succ}\ (\mathsf{snd}\ \langle\mathsf{nat}\rangle\ \langle\mathsf{nat}\rangle\ \mathsf{p})))$

$$\mathbf{pred} = \lambda \mathsf{n}{:}\mathsf{nat}.\ \mathsf{fst}\ \langle\mathsf{nat}\rangle\ \langle\mathsf{nat}\rangle\ (\mathsf{n}\ \langle\mathsf{output}\rangle\ \mathsf{step}\ \mathsf{base})$$

Exercise: define the **factorial** function using iteration over pairs (in a way similar to **pred**).

Algebraic datatypes: lists

It is possible to represent any inductively defined algebraic datatype in λ_2. We present the method by means of a concrete example.

Consider the datatype *lists of naturals*, written below in Haskell notation:

```
data ListNat = Empty
             | Cons Nat ListNat
```

The datatype `listNat` can be represented by the polymorphic type

$$\forall C.\ C \to (\mathsf{nat} \to C \to C) \to C$$

Notice that each argument before the final C correspond to the signature of each constructor, assuming $C = \texttt{listNat}$.

$$\forall C.\ \overbrace{C}^{\mathsf{Empty}} \to \overbrace{(\mathsf{nat} \to C \to C)}^{\mathsf{Cons}} \to C$$

Algebraic datatypes: lists (2)

The constructors $\mathsf{empty} : \mathsf{listNat}$ and $\mathsf{cons} : \mathsf{nat} \to \mathsf{listNat} \to \mathsf{listNat}$ are represented by the following terms:

$$\begin{aligned}
\mathsf{empty} &: \forall C.\ C \to (\mathsf{nat} \to C \to C) \to C \\
\mathsf{empty} &= \Lambda C.\ \lambda e{:}C.\ \lambda c{:}(\mathsf{nat} \to C \to C).\ e \\
\mathsf{cons} &: \mathsf{nat} \to (\forall C.\ C \to (\mathsf{nat} \to C \to C) \to C) \\
&\qquad \to (\forall C.\ C \to (\mathsf{nat} \to C \to C) \to C) \\
\mathsf{cons} &= \lambda n{:}\mathsf{nat}.\ \ \lambda \ell{:}(\forall C.\ C \to (\mathsf{nat} \to C \to C) \to C). \\
&\qquad \Lambda C.\ \lambda e{:}C.\ \lambda c{:}(\mathsf{nat} \to C \to C).\ c\ n\ (\ell\ \langle C\rangle\ e\ c)
\end{aligned}$$

Notice that the polymorphic definitions of list constructors in λ_2 resemble the Scott encoding of lists in λ. The main distinction is that the recursive arguments (ℓ in the case of cons) must be fully instantiated/filled with arguments in order to obtain something of the required type C. The end result is actually a Church-style encoding.

Algebraic datatypes: lists (3)

Objects of type $\forall \mathsf{C}.\ \mathsf{C} \to (\mathsf{nat} \to \mathsf{C} \to \mathsf{C}) \to \mathsf{C}$ can receive as arguments

- an output type O
- a value v of type O (the empty case)
- a function f of type $\mathsf{nat} \to \mathsf{O} \to \mathsf{O}$ (the cons case)

and generate a sequence of calls to f, finalized by v, following the list structure.

For example, let ℓ the be list constructed as

$$\ell = \mathsf{cons}\ \mathbf{4}\ (\mathsf{cons}\ \mathbf{2}\ (\mathsf{cons}\ \mathbf{5}\ \mathsf{empty}))$$

and let $\mathsf{O} = \mathsf{nat}$, $\mathsf{v} = \mathbf{0}$ and $\mathsf{f} = \mathbf{add}$. Therefore,

$$\ell\ \langle \mathsf{nat} \rangle\ \mathbf{0}\ \mathbf{add} =_{\beta} \mathbf{add}\ \mathbf{4}\ (\mathbf{add}\ \mathbf{2}\ (\mathbf{add}\ \mathbf{5}\ \mathbf{0})) =_{\beta} \mathbf{11}$$

This means we can *apply* the constructed structure as an **iterator**, which works similarly to a Church-encoded natural in λ.

Algebraic datatypes: lists (4)

In λ_2 we need to use **list iterators** to define both *tests* and *destructors*.

The operations isEmpty, isCons and head are easy to define, as shown below:

$$
\begin{array}{lcl}
\mathsf{isEmpty} & : & \mathsf{listNat} \to \mathsf{bool} \\
\mathsf{isEmpty} & = & \lambda\ell{:}\mathsf{listNat}.\ \ell\ \langle\mathsf{bool}\rangle\ \mathbf{true}\ (\lambda \mathsf{n}{:}\mathsf{nat}.\ \lambda \mathsf{k}{:}\mathsf{bool}.\ \mathbf{false}) \\
\\
\mathsf{isCons} & : & \mathsf{listNat} \to \mathsf{bool} \\
\mathsf{isCons} & = & \lambda\ell{:}\mathsf{listNat}.\ \ell\ \langle\mathsf{bool}\rangle\ \mathbf{false}\ (\lambda \mathsf{n}{:}\mathsf{nat}.\ \lambda \mathsf{k}{:}\mathsf{bool}.\ \mathbf{true}) \\
\\
\mathsf{head} & : & \mathsf{listNat} \to \mathsf{nat} \\
\mathsf{head} & = & \lambda\ell{:}\mathsf{listNat}.\ \ell\ \langle\mathsf{nat}\rangle\ \mathbf{0}\ (\lambda \mathsf{n}{:}\mathsf{nat}.\ \lambda \mathsf{k}{:}\mathsf{nat}.\ \mathsf{n})
\end{array}
$$

Notice: for simplicity, head empty returns $\mathbf{0}$

Algebraic datatypes: lists (5)

The definition of tail is trickier: we need to employ a strategy similar to the one used for defining pred in Church naturals (in λ):

- use pairs of listNat to iterate over the input list. We denote pairL, fstL and sndL the pair operations instantiated on type listNat × listNat;
- at each cons, we keep the previous list at the left of the pair, and reconstruct the original list at the right of the pair;
- for simplicity, tail empty returns empty.

$$\begin{array}{lcl}
\mathsf{shiftCons} & : & \mathsf{nat} \to (\mathsf{listNat} \times \mathsf{listNat}) \to (\mathsf{listNat} \times \mathsf{listNat}) \\
\mathsf{shiftCons} & = & \lambda \mathsf{n}\,{:}\,\mathsf{nat}.\ \lambda \mathsf{p}\,{:}\,\mathsf{listNat} \times \mathsf{listNat}.\ \mathsf{pairL}\ \ (\mathsf{sndL\ p})\ \ (\mathsf{cons\ n\ (sndL\ p)}) \\
 & & \\
\mathsf{tail} & : & \mathsf{listNat} \to \mathsf{listNat} \\
\mathsf{tail} & = & \lambda \ell\,{:}\,\mathsf{listNat}.\ \mathsf{fstL}\ (\ell\ \langle \mathsf{listNat} \times \mathsf{listNat} \rangle\ (\mathsf{pairL\ empty\ empty})\ \mathsf{shiftCons})
\end{array}$$

Algebraic datatypes: lists (6)

We can also pass the type of the list contents as an extra parameter for the constructors (instead of having a fixed type nat):

$$\mathsf{list}_A = \forall C.\, C \to (A \to C \to C) \to C$$

$$\begin{aligned}
\mathsf{empty} \;&:\; \forall A.\, \forall C.\, C \to (A \to C \to C) \to C \\
\mathsf{empty} \;&=\; \Lambda A.\, \Lambda C.\, \lambda e{:}C.\, \lambda c{:}(A \to C \to C).\, e \\
\mathsf{cons} \;&:\; \forall A.\, A \to (\forall C.\, C \to (A \to C \to C) \to C) \\
&\qquad\quad \to (\forall C.\, C \to (A \to C \to C) \to C) \\
\mathsf{cons} \;&=\; \Lambda A.\, \lambda a{:}A.\; \lambda \ell{:}(\forall C.\, C \to (A \to C \to C) \to C). \\
&\qquad\quad \Lambda C.\, \lambda e{:}C.\, \lambda c{:}(A \to C \to C).\, c\; a\; (\ell\; \langle C \rangle\; e\; c)
\end{aligned}$$

Example: $\mathsf{cons}\ \langle \mathsf{nat} \rangle\ 5\ (\mathsf{cons}\ \langle \mathsf{nat} \rangle\ 2\ (\mathsf{empty}\ \langle \mathsf{nat} \rangle))$

In λ_2 it is not possible to represent the operation $\mathsf{list} : \star \to \star$.

Empty and Unit datatypes

The smallest algebraic datatypes are

- an algebraic datatype `Empty`, without any constructors, and
- an algebraic datatype `Unit`, with one parameterless constructor.

```
data Empty =
data Unit  = Single
```

By using the encoding we have seen, we obtain:

$$\begin{aligned} \mathsf{empty} &= \forall C.\, C \\ \mathsf{unit} &= \forall C.\, C \to C \\ \mathsf{single} &= \Lambda C.\, \lambda c\!:\!C.\, c \end{aligned}$$

Notice that unit is the type of the polymorphic identity, which happens to be exactly the constructor single.

The type empty is not inhabitted in λ_2. A constructor of type empty would be able to obtain, given any type C, a particular element of this type.

Properties of $\lambda 2$

We review below some important properties of λ_2 (à la Church). We will not **prove** them here, because

- some are straightforward extensions of proofs of previous calculi, i.e. substitution lemma, progress, preservation.
- some can be quite tricky, i.e. strong normalization

Uniqueness of types: if $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$ then $\tau = \tau'$

Substitution lemma: if $\Gamma, x : \tau_1 \vdash e : \tau$ and $\Gamma \vdash e_1 : \tau_1$ then $\Gamma \vdash [e_1/x]e : \tau$

Preservation: if $\Gamma \vdash e : \tau$ and $e \to_\beta e'$ then $\Gamma \vdash e' : \tau$

Progress: if $\Gamma \vdash e : \tau$, then either $\mathsf{NF}(e)$ or exists e' such that $e \to e'$

Properties of λ_2 (2)

Subject Reduction is a consequence of progress and preservation.

Church-Rosser: if $\Gamma \vdash e : \tau$, $e \twoheadrightarrow_\beta e_1$ and $e \twoheadrightarrow_\beta e_2$, then exists e' such that $e_1 \twoheadrightarrow e'$ and $e_2 \twoheadrightarrow e'$.

Strong Normalization: (Girard, 1972) if $\Gamma \vdash e : \tau$, then $e \Downarrow$

The type reconstruction problem: given a closed untyped term $e \in \Lambda$, is there a well-typed term $e' \in \lambda_2$ such that $|e'| = e$?

Undecidability of type reconstruction: (Wells, 1994) the type reconstruction problem for λ_2 is *undecidable*.

Properties of λ_2 (3)

In a popular article of 1989, Philip Wadler coined the expression *"free theorems"*.

The idea is that each **polymorphic type** τ induces some properties (automatically) on every term of type τ.

For instance, consider the type $\forall A.\mathsf{List}(A) \to \mathsf{List}(A)$. Notice that this is the type of the $\mathsf{reverse}$ and the tail operations on lists (assuming $\mathsf{tail\ empty} = \mathsf{empty}$).

Assume $f : \forall A.\mathsf{List}(A) \to \mathsf{List}(A)$. The following theorem holds for f, and follows from its type alone: for each type X, type Y and total function $g : X \to Y$, we have

$$(\mathsf{map}\ g) \circ (f\ \langle X \rangle) = (f\ \langle Y \rangle) \circ (\mathsf{map}\ g)$$

Exercise: test the property above with $\mathsf{reverse}$ and tail.

This notion of *free theorems* is actually a consequence of a (quite technical) result from Reynolds, referred as the *abstraction theorem* (Reynolds 1983).

Variation of λ_2: Hindley-Milner polymorphism

The fact that the type reconstruction problem is undecidable for λ_2 restricts the usefulness of λ_2 as basis for the type system of general purpose languages with **type inference**.

However, if we restrict the types of λ_2 to the format

$$\forall \vec{X}.\sigma$$

where σ is a type without $\forall$ bindings (simple type), we obtain a fragment of λ_2 with decidable type reconstruction.

This kind of polymorphism is called **let-polymorphism**, **ML-style polymorphism** or **Hindley-Milner** type system, and it is the basis for the polymorphic type system in many functional programming languages such as Ocaml and Haskell.

Content

Higher-order Polymorphic Lambda Calculus

Type constructors

Recall λ_2: given two types $\tau_1, \tau_2 \in \mathbb{T}_2$, the polymophic type $\forall C.(\tau_1 \to \tau_2 \to C) \to C$ represents the product type $\tau_1 \times \tau_2$.

For pairs we can define a *generic constructor* pair and *generic projections* fst and snd that receive arbitrary types τ_1 and τ_2 as arguments.

However, in λ_2 we **do not have** a generic type-level operation

$$\mathsf{product} : \star \to \star \to \star$$

(usually written _ $\times$ _) that, given two types τ_1 and τ_2, construct for us the type $\tau_1 \times \tau_2 = \forall C.(\tau_1 \to \tau_2 \to C) \to C$

In other words, λ_2 lacks **types depending on types**, also known as **type constructors**.

Type constructors (2)

Statically-typed functional programming languages usually support type constructors for many useful data structures such as pairs, lists, optional types (Maybe), disjoint unions (Either), etc.

For example, consider the definition of the (parameterized) datatype Maybe:

```
data Maybe a = Nothing | Just a
```

We consider Maybe Int, Maybe Bool, Maybe (Maybe Int) as “complete types”.

However, just Maybe by itself is a *“type-wannabe”* (or a type pattern): it requires a type argument to become a “complete” type, with well-defined values.

The “type” of Maybe is $\star \to \star$, which means that it receives a type $\tau : \star$ as input, and return the type $(\text{Maybe}\ \tau) : \star$ as output. The $\star$ symbol is a synonym for the “type” of “complete types”.

Type constructors (3)

We intend to introduce the concept of **types that depend on types**.

Starting with λ_2 we will define two new syntatic forms *at the type level*:

- a type abstraction $\lambda X : \kappa . \tau$
- a type application $\tau_1\ \tau_2$,

resulting in the **higher-order polymorphic lambda calculus** (HOPLC), abbreviated as $\lambda\omega$ (lambda omega). It is also known as **System F**ω.

The κ (greek letter kappa) in the lambda binding refers to a particular *type of type constructors*. For the lack of a better word, these "types of types" are referred as **kinds**.

Notice: in λ_ω, the lowercase lambda $\lambda ? : ? . ?$ will refer to an abstraction at the same level on the input and output (term or type), while the uppercase lambda $\Lambda X : \kappa . e$ will refer to an upper-level abstraction (i.e. type as input, term as output).

Kinds

The set $\mathbb{K}$ of **kinds** is defined by the following syntax:

$$\begin{array}{lcl} \kappa & \in & \mathbb{K} \\ \kappa & ::= & \star \mid \kappa_1 \to \kappa_2 \end{array}$$

Example: (of kinds)

- $\star$
- $\star \to \star$
- $\star \to (\star \to \star)$
- $(\star \to \star) \to (\star \to \star)$ …

The $\to$ constructor (at kind level) is right-associative, i.e:

$$\star \to \star \to \star \;=\; \star \to (\star \to \star)$$

Type constructors

The set $\mathbb{T}_\omega$ of **type constructors** is defined by the following syntax:

$$\begin{array}{lcll} X & \in & \mathsf{TypeVar} & \\ \tau & \in & \mathbb{T}_\omega & \\ \tau & ::= & X & \\ & | & \tau_1 \rightarrow \tau_2 & \\ & | & \forall X{:}\kappa.\tau & \text{(kind signature added)} \\ & | & \lambda X{:}\kappa.\tau & \text{(type abstraction)} \\ & | & \tau_1\ \tau_2 & \text{(type application)} \end{array}$$

Notice:

- type variables in declarations are now always annotated with a particular kind.
- the type $\forall X.\tau$ in λ_2 is equivalent to the type constructor $\forall X{:}\star.\tau$ in λ_ω.
- we assume equality of types is actually α-equivalence (i.e. we allow the renaming of bound type variables). The definition is as usual, and we will not make it explicit here.

Types and proper constructors

We will eventually define a type system that associates a type constructor to a kind (under some context), which we will write $\Gamma \vdash \tau : \kappa$.

We employ a specific *nomenclature* for type constructors, based on their respective kinds:

- we will reserve the term **type** only for type constructors of kind $\star$, i.e. what we have been calling a “complete type”.
- we will use the expression **proper constructor** to denote a type constructor whose kind is **not** $\star$, i.e. an “incomplete type”.
- we use **type constructor** for both types and proper constructors.
- the metavariable τ ranges over all type constructors.

Terms

Definition: The set Λ_ω of terms is defined by the following abstract grammar:

$$\begin{array}{lcl}
x & \in & \mathsf{Var} \\
e & \in & \Lambda_\omega \\
e & ::= & x \\
 & | & e_1\ e_2 \\
 & | & \lambda x{:}\tau.e_1 \\
 & | & \Lambda X{:}\kappa.e_1 \\
 & | & e_1\ \langle\tau\rangle
\end{array}$$

Notice:

- the only distinction between terms in λ_2 and λ_ω is the presence of *kind annotations* for type variables in type abstractions.
- a term $\Lambda X.e_1$ in λ_2 is represented by the term $\Lambda X{:}\star.e_1$ in λ_ω.
- we assume that term equality is α-equivalence, i.e. we allow the (non-capturing) renaming of both variables and type variables in terms.

Levels

In $\lambda_{\rightarrow}$ and λ_2 there are terms and types. λ_ω introduces kinds (types of types).

We will use the symbol $\square$ to denote the (unique) *type of kinds*, or "superkind".
This is helpful only for syntactic reasons, since all kinds $\kappa \in \mathbb{K}$ are well-formed.
This superkind is unique, and will serve as its own meta-variable.

We have the following **four levels** in λ_ω, with their respective meta-variables.

name	**terms**	**type constructors**	**kinds**	**superkinds**
set	Λ_ω	$\mathbb{T}_\omega$	$\mathbb{K}$	$\{\square\}$
metavar	e	τ	κ	$\square$
examples	$\Lambda X{:}\star.\ \lambda x{:}X.\ x$ $\lambda x{:}(\forall X.X).\ x$ $(\lambda y{:}Y \to Z.\ u)\ a$	$X \to Y$ $\forall X{:}\star.X \to X$ $(\lambda X{:}\star.X)\ Y$	$\star$ $(\star \to \star) \to \star$ $\star \to \star$	$\square$

Therefore, there are three possible placements for the ":" of the type relation:

$$e \overset{(1)}{:} \tau \overset{(2)}{:} \kappa \overset{(3)}{:} \square$$

Sorts

We define **the set of sorts** as $\mathbb{S} = \{\star, \square\}$, containing:

- $\star$, the kind of types
- $\square$, the unique superkind

The meta-variable $\mathsf{s} \in \mathbb{S}$ will be used whenever an arbitrary sort is required.

Sorts are relevant because they affect declarations of *variables* and *type variables*:

- variable declarations require a **type** (of sort $\star$).
- type variable declarations require a **kind** (of sort $\square$).

For example, a declaration $\mathsf{x} : \tau$ where $\tau : \star \to \star$ will **not be allowed**, since τ needs to be a **type** (not a proper constructor).

Contexts

Contexts in λ_ω are similar to the ones in λ_2, with the following remarks

- variables may only be associated with types
- type variables may be associated with any kind
- types may contain abstractions and applications

Example: $X{:}\star,\ x{:}X \to X,\ L{:}\star \to \star,\ z{:}((\lambda U{:}\star.\ L\ U)\ X)$

We could **try** to define CTX as the smallest set such that:

1. $\bullet \in \mathsf{CTX}$
2. if $\Gamma \in \mathsf{CTX}$, $X \notin \mathsf{dom}(\Gamma)$ and $\Gamma \vdash \kappa : \Box$, then $(\Gamma, X : \kappa) \in \mathsf{CTX}$
3. if $\Gamma \in \mathsf{CTX}$, $x \notin \mathsf{dom}(\Gamma)$ and $\Gamma \vdash \tau : \star$, then $(\Gamma, x : \tau) \in \mathsf{CTX}$

However we would need to query the type system at levels (2) and (3) to extend the context, and these would required well-formed contexts. For convenience, the construction of contexts will be **included** in the type system rules.

Type rules (level 3) kinds:superkind

Let Env be the set of objects defined by the following grammar:

$$\Gamma ::= \bullet \mid \Gamma', x : \tau \mid \Gamma', X : \kappa$$

$$(\_ \vdash \_ : \_) \subseteq \mathsf{Env} \times \mathbb{K} \times \{\Box\}$$

$$\frac{}{\bullet \vdash \star : \Box} \quad (\textsc{sort})$$

$$\frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma \vdash \kappa_2 : \Box}{\Gamma \vdash \kappa_1 \to \kappa_2 : \Box} \quad (\textsc{formKind})$$

$$\frac{\Gamma \vdash \kappa : \Box \qquad X \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \kappa' : \Box}{\Gamma, X : \kappa' \vdash \kappa : \Box} \quad (\textsc{TVarWeakKind})$$

$$\frac{\Gamma \vdash \kappa : \Box \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \tau' : \star}{\Gamma, x : \tau' \vdash \kappa : \Box} \quad (\textsc{VarWeakKind})$$

Type rules (level 2) types:kinds

$$(\_ \vdash \_ : \_) \subseteq \mathsf{Env} \times \mathbb{T}_\omega \times \mathbb{K}$$

$$\frac{X \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \kappa : \square}{\Gamma, X{:}\kappa \vdash X : \kappa}\ (\textsc{typeVar})$$

$$\frac{\Gamma \vdash \tau_1 : \kappa \to \kappa' \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \tau_1\ \tau_2 : \kappa'}\ (\textsc{appType})$$

$$\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \to \tau_2 : \star}\ (\textsc{formType})$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad X \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \kappa' : \square}{\Gamma, X{:}\kappa' \vdash \tau : \kappa}\ (\textsc{TVarWeakType})$$

$$\frac{\Gamma, X{:}\kappa \vdash \tau : \star}{\Gamma \vdash \forall X{:}\kappa.\tau : \star}\ (\textsc{forallType})$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau' : \star}{\Gamma, x{:}\tau' \vdash \tau : \kappa}\ (\textsc{VarWeakType})$$

$$\frac{\Gamma, X{:}\kappa \vdash \tau : \kappa'}{\Gamma \vdash \lambda X{:}\kappa.\tau : \kappa \to \kappa'}\ (\textsc{lambdaType})$$

Type rules (level 1) terms:types

$$(\_ \vdash \_ : \_) \subseteq \mathsf{Env} \times \Lambda_\omega \times \mathbb{T}_\omega$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \tau : \star}{\Gamma, x{:}\tau \vdash x : \tau}\ (\textsc{var})$$

$$\frac{\Gamma \vdash e : \tau \qquad X \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \kappa' : \square}{\Gamma, X{:}\kappa' \vdash e : \tau}\ (\textsc{TVarWeakTerm})$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'}\ (\textsc{lambdaTerm})$$

$$\frac{\Gamma \vdash e : \tau \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \tau' : \star}{\Gamma, x{:}\tau' \vdash e : \tau}\ (\textsc{VarWeakTerm})$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}\ (\textsc{appTerm})$$

$$\frac{\Gamma, X{:}\kappa \vdash e : \tau}{\Gamma \vdash \Lambda X{:}\kappa.e : \forall X{:}\kappa.\tau}\ (\textsc{typeAbsTerm})$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 =_\beta \tau_2 \qquad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash e : \tau_2}\ (\textsc{conversion})$$

$$\frac{\Gamma \vdash e : \forall X{:}\kappa.\tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\ \langle\tau\rangle : [\tau/X]\tau'}\ (\textsc{typeAppTerm})$$

Beta-equivalence on types

The type system rule CONVERSION says that a term may have many, distinct types, as long as all the types are well-formed (at the context) and β-equivalent.

Example:

- $\Gamma \vdash \mathbf{5} : \mathsf{nat}$
- $\Gamma \vdash \mathbf{5} : (\lambda X{:}\star.\mathsf{nat})\ \mathsf{bool}$
- $\Gamma \vdash \mathbf{5} : (\lambda X{:}\star.X)\ \mathsf{nat}$

Here we will define the relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ on **types** first, and after we define $=_\beta$ in a way similar to the STLC $\lambda_\rightarrow$.

Notice: $=_\beta$ can also be defined directly without previously defining a reduction $\rightarrow_\beta$, based on a collection of identities.

Substitution and redexes on types

As in λ_2, there are three substitution operations in λ_ω:

- [type / type variable] type
- [type / type variable] term
- [term / variable] term

We need to extend the definition of the first kind of substitution to account for lambdas and applications at the type level (again, here we assume the Barendregt Assumption).

$$\begin{array}{lcl} [\tau/Y](\lambda X{:}\kappa.\tau_1) & = & \lambda X{:}\kappa.[\tau/Y]\tau_1 \\ [\tau/Y](\tau_1\ \tau_2) & = & ([\tau/Y]\tau_1)\ ([\tau/Y]\tau_2) \end{array}$$

A **redex** within a type is an inner type in the format $(\lambda X{:}\kappa.\tau)\ \tau'$. Its respective contractum is $[\tau'/X]\tau$.

Reduction on types

The beta-reduction relation on types is the smallest relation $\rightarrow_\beta \subseteq \mathbb{T}_\omega \times \mathbb{T}_\omega$ satisfying:

$$\frac{}{(\lambda X{:}\kappa.\tau)\,\tau' \rightarrow_\beta [\tau'/X]\tau}\ (\beta) \qquad \frac{\tau \rightarrow_\beta \tau'}{\lambda X{:}\kappa.\tau \rightarrow_\beta \lambda X{:}\kappa.\tau'}\ (\textsc{cptLam})$$

$$\frac{\tau_1 \rightarrow_\beta \tau_1'}{(\tau_1 \rightarrow \tau_2) \rightarrow_\beta (\tau_1' \rightarrow \tau_2)}\ (\textsc{cptForm1}) \qquad \frac{\tau_1 \rightarrow_\beta \tau_1'}{\tau_1\ \tau_2 \rightarrow_\beta \tau_1'\ \tau_2}\ (\textsc{cptApp1})$$

$$\frac{\tau_2 \rightarrow_\beta \tau_2'}{(\tau_1 \rightarrow \tau_2) \rightarrow_\beta (\tau_1 \rightarrow \tau_2')}\ (\textsc{cptForm2}) \qquad \frac{\tau_2 \rightarrow_\beta \tau_2'}{\tau_1\ \tau_2 \rightarrow_\beta \tau_1\ \tau_2'}\ (\textsc{cptApp2})$$

- $\twoheadrightarrow_\beta$ is the reflexive, transitive closure of $\rightarrow_\beta$, and $=_\beta$ is the smallest relation satisfying: if $\tau_1 \twoheadrightarrow_\beta \tau$ and $\tau_2 \twoheadrightarrow_\beta \tau$, then $\tau_1 =_\beta \tau_2$.
- the beta-reduction on well-formed types is *strongly normalizing* and Church-Rosser.
- to determine $\tau_1 =_\beta \tau_2$ we can just simplify both terms and compare their normal forms (which is required for type checking due to the CONVERSION rule).

Semantics

The semantics of λ_ω is quite similar to λ_2, with the main distinction being the inclusion of *reduction on types*.

Beta reduction $(\_ \rightarrow_\beta \_) \subseteq \Lambda_\omega \times \Lambda_\omega$ is the smallest relation satisfying:

$$\frac{}{(\lambda x : \tau . e)\ e' \rightarrow_\beta [e'/x]e}\ (\beta) \qquad \frac{e \rightarrow_\beta e'}{\lambda x : \tau . e \rightarrow_\beta \lambda x : \tau . e'}\ (\textsc{cpLam2}) \qquad \frac{e \rightarrow_\beta e'}{\Lambda X : \kappa . e \rightarrow_\beta \Lambda X : \kappa . e'}\ (\textsc{cpTyLam})$$

$$\frac{}{(\Lambda X : \kappa . e)\ \langle \tau \rangle \rightarrow_\beta [\tau/X]e}\ (\textsc{inst}) \qquad \frac{e_1 \rightarrow_\beta e_1'}{e_1\ e_2 \rightarrow_\beta e_1'\ e_2}\ (\textsc{cpApp1}) \qquad \frac{e \rightarrow_\beta e'}{e\ \langle \tau \rangle \rightarrow_\beta e'\ \langle \tau \rangle}\ (\textsc{cpTyApp1})$$

$$\frac{\tau \rightarrow_\beta \tau'}{\lambda x : \tau . e \rightarrow_\beta \lambda x : \tau' . e}\ (\textsc{cpLam1}) \qquad \frac{e_2 \rightarrow_\beta e_2'}{e_1\ e_2 \rightarrow_\beta e_1\ e_2'}\ (\textsc{cpApp2}) \qquad \frac{\tau \rightarrow_\beta \tau'}{e\ \langle \tau \rangle \rightarrow_\beta e\ \langle \tau' \rangle}\ (\textsc{cpTyApp2})$$

Notice: this reduction on types could, in principle, be ignored by the semantics, given that type normalization is used primarily for type checking.

Properties of λ_ω

The λ_ω formalism shares many properties with λ_2, including:

- Subject reduction
- Church-Rosser property
- Strong normalization (for terms)
- Undecidable type reconstruction
- Existence of well-formed, non-inhabited types

The type language of λ_ω has a notion of beta-reduction which satisfies

- Strong normalization
- Church-Rosser property

We will not explore in detail the meta-theory of λ_ω (for the same reasons explained when we discussed λ_2).

Programming in λ_ω

All the encodings seen for λ_2 are valid in λ_ω: booleans, naturals, pairs, lists, unit and empty.

The main improvement is that we now can use *proper constructors* to represent type-level operations as, for example:

$$\begin{array}{lcl} \mathsf{Product} & : & \star \to \star \to \star \\ \mathsf{Product} & = & \lambda A{:}\star.\, \lambda B{:}\star.\, (\forall C.\, (A \to B \to C) \to C) \\ \\ \mathsf{List} & : & \star \to \star \\ \mathsf{List} & = & \lambda A{:}\star.\, (\forall C.\, C \to (A \to C \to C) \to C) \end{array}$$

Example:

- $A \times B = \mathsf{Product}\ A\ B$
- $\mathsf{list}_A = \mathsf{List}\ A$

We will explore now a few more datatypes and their respective type constructors.

Maybe datatype

The definition of the Maybe datatype in Haskell is

```
data Maybe a = Nothing | Just a
```

Given a type A, the elements of Maybe A are

- Nothing, representing *undefined* or *failure*;
- Just x, representing a successful outcome with value x (of type A).

We have the following encoding of maybe in λ_ω:

$$\begin{array}{lcl} \mathsf{Maybe} & : & \star \to \star \\ \mathsf{Maybe} & = & \lambda A{:}\star.\ \forall C.\ C \to (A \to C) \to C \\ \\ \mathsf{nothing} & = & \Lambda A{:}\star.\ \Lambda C{:}\star.\ \lambda n{:}C.\ \lambda j{:}(A \to C).\ n \\ \mathsf{just} & = & \Lambda A{:}\star.\ \lambda a{:}A.\ \Lambda C{:}\star.\ \lambda n{:}C.\ \lambda j{:}(A \to C).\ j\ a \end{array}$$

Maybe datatype: destructors

In Haskell, it is usual to employ *pattern matching* over values of type Maybe a:

```
foo    :  Maybe Nat -> Nat
foo m  =  case m of
            Nothing -> 0
            Just x  -> x+1
```

Similarly to booleans, naturals and lists, we can extract information from a term of type Maybe τ by passing adequate arguments to it:

Example:

$$\begin{array}{lcl} \mathsf{foo} & : & \mathsf{Maybe\ Nat} \to \mathsf{Nat} \\ \mathsf{foo} & = & \lambda \mathsf{m} : \mathsf{Maybe\ Nat}.\ \mathsf{m}\ \langle \mathsf{Nat} \rangle\ 0\ (\lambda \mathsf{x} : \mathsf{Nat}.\mathbf{succ}\ \mathsf{x}) \end{array}$$

Exercise: define a generic destructor fromMaybe with the signature:

$$\mathsf{fromMaybe} : \forall \mathsf{A} : \star.\ \forall \mathsf{C} : \star.\ (\mathsf{Maybe\ A}) \to \mathsf{C} \to (\mathsf{A} \to \mathsf{C}) \to \mathsf{C}$$

Safe list operations with Maybe

When considering List Nat, for simplicity we assumed

- head empty $=$ $\mathbf{0}$
- tail empty $=$ empty

Notice that we specified a *default* value $\mathbf{0}$ for head empty, which is not possible when defining the list operations in a generic way.

Is is arguably better to define safe, generic versions of the list operations using the following signatures:

- head : $\forall$A:$\star$. List A $\rightarrow$ Maybe A
- tail : $\forall$A:$\star$. List A $\rightarrow$ Maybe (List A)

These operations return Nothing when trying head empty or tail empty, affirming that an error has occurred.

Exercise: implement these versions of head and tail in λ_ω.

Either datatype

The definition of the Either datatype in Haskell is

```
data Either a b = Left a | Right b
```

Given types A and B, the elements of $\mathsf{Either\ A\ B}$ are

- $\mathsf{Left\ a}$, representing an element a of type A
- $\mathsf{Right\ b}$, representing an element b of type B

We have the following encoding of Either in λ_ω:

$$\begin{array}{lcl}
\mathsf{Either} & : & \star \to \star \to \star \\
\mathsf{Either} & = & \lambda \mathsf{A}{:}\star.\ \lambda \mathsf{B}{:}\star.\ \forall \mathsf{C}.\ (\mathsf{A} \to \mathsf{C}) \to (\mathsf{B} \to \mathsf{C}) \to \mathsf{C} \\
\\
\mathsf{left} & : & \forall \mathsf{A}{:}\star.\ \forall \mathsf{B}{:}\star.\ \mathsf{A} \to \mathsf{Either\ A\ B} \\
\mathsf{left} & = & \Lambda \mathsf{A}{:}\star.\ \Lambda \mathsf{B}{:}\star.\ \lambda \mathsf{a}{:}\mathsf{A}.\ \Lambda \mathsf{C}{:}\star.\ \lambda \mathsf{l}{:}(\mathsf{A} \to \mathsf{C}).\ \lambda \mathsf{r}{:}(\mathsf{B} \to \mathsf{C}).\ \mathsf{l\ a} \\
\\
\mathsf{right} & : & \forall \mathsf{A}{:}\star.\ \forall \mathsf{B}{:}\star.\ \mathsf{B} \to \mathsf{Either\ A\ B} \\
\mathsf{right} & = & \Lambda \mathsf{A}{:}\star.\ \Lambda \mathsf{B}{:}\star.\ \lambda \mathsf{b}{:}\mathsf{B}.\ \Lambda \mathsf{C}{:}\star.\ \lambda \mathsf{l}{:}(\mathsf{A} \to \mathsf{C}).\ \lambda \mathsf{r}{:}(\mathsf{B} \to \mathsf{C}).\ \mathsf{r\ b}
\end{array}$$

Either datatype: destructor

The destructor of the $\mathsf{Either\ A\ B}$ datatype is a case expression

$$\mathsf{case}(\mathsf{e}, \mathsf{f}, \mathsf{g})$$

that, given $\mathsf{e} : \mathsf{Either\ A\ B}$ and two functions $\mathsf{f} : \mathsf{A} \to \mathsf{U}$ and $\mathsf{g} : \mathsf{B} \to \mathsf{U}$, it applies or f or g to the *argument* of e to obtain a result of type U:

- if $\mathsf{e} = \mathsf{Left\ a}$ then $\mathsf{case}(\mathsf{e}, \mathsf{f}, \mathsf{g}) = \mathsf{f\ a}$
- if $\mathsf{e} = \mathsf{Right\ b}$ then $\mathsf{case}(\mathsf{e}, \mathsf{f}, \mathsf{g}) = \mathsf{g\ b}$

We have the following generic encoding of case in λ_ω:

$$\begin{aligned}
\mathsf{case} \quad &: \quad \forall \mathsf{A}.\ \forall \mathsf{B}.\ \forall \mathsf{U}.\ \mathsf{Either\ A\ B} \to (\mathsf{A} \to \mathsf{U}) \to (\mathsf{B} \to \mathsf{U}) \to \mathsf{U} \\
\mathsf{case} \quad &= \quad \Lambda \mathsf{A}.\ \Lambda \mathsf{B}.\ \Lambda \mathsf{U}.\ \lambda \mathsf{x} : \mathsf{Either\ A\ B}.\ \lambda \mathsf{f} : \mathsf{A} \to \mathsf{U}.\ \lambda \mathsf{g} : \mathsf{B} \to \mathsf{U}.\ \mathsf{x}\ \langle \mathsf{U} \rangle\ \mathsf{f}\ \mathsf{g}
\end{aligned}$$

Programming languages and λ_ω

Although λ_ω is not Turing-complete (since it is strongly normalizing), many relevant total functions are representable in the pure calculus (for instance, factorial).

Notice: extensions / variations of the concepts present in λ_ω occur in *modern Turing-complete programming languages*.

Example: the **ML family** of languages (which include Standard ML, Ocaml and F#) have both *polymorphism* (although slightly restricted) and *type constructors*, and include:

- **primitive datatypes** (naturals, booleans, strings)
- **fixpoint operators** (`let rec`)
- a **deterministic** reduction relation (call by value)
- another notion of normal form: **weak head normal form**, i.e. lambda expressions are considered values and evaluation does not occur under them.

Encoding IPL in $\lambda\omega$

As mentioned in previous lectures, each of the typed variations of lambda calculi we have seen ($\lambda_{\rightarrow}$, λ_2 and λ_ω) correspond to a particular deductive system for some intuitionistic logic.

Depending on which constructions are available (pairs, disjoint unions, bottom type) in each lambda calculus, we have a distinct (intuitionistic) logic associated with it.

In particular, the logics of λ_2 and λ_ω have second-order elements (quantification over types).

Instead of exploring these logics individually, we will focus here on the previously presented **intuitionistic propositional logic** (IPL), and show how it can be encoded in the **pure** version of λ_ω.

This will allow us to use λ_ω to build proofs for IPL propositions, i.e. we will use λ_ω as a **proof assistant** for IPL.

Intuitionistic Propositional Logic (review)

Syntax:

$$\begin{array}{lll} \mathsf{P} & \in & \mathsf{Literals} \\ \varphi & ::= & \mathsf{P} \mid \bot \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \\ \Gamma & ::= & \{\} \mid \Gamma_1, \varphi \end{array} \qquad (\Gamma \text{ seen as a set})$$

Deductive system:

$$\frac{\varphi \in \Gamma}{\Gamma \vdash \varphi}\ (\text{Ax}) \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}\ (\Rightarrow\text{-I})$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot\text{-E}) \qquad \frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\Rightarrow\text{-E})$$

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (\wedge\text{-I}) \qquad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ (\vee\text{-I1})$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ (\wedge\text{-E1}) \qquad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \psi \vee \varphi}\ (\vee\text{-I2})$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\ (\wedge\text{-E2}) \qquad \frac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma \vdash \varphi \Rightarrow \xi \qquad \Gamma \vdash \psi \Rightarrow \xi}{\Gamma \vdash \xi}\ (\vee\text{-E})$$

Encoding formulas

$$\_' : \mathsf{Formula} \to \mathbb{T}_\omega$$

$\mathsf{P}' = \mathsf{P}$ (literals are represented by *type variables*)

$\bot' = \forall \mathsf{C}{:}\star.\mathsf{C}$

$(\varphi_1 \Rightarrow \varphi_2)' = \varphi_1' \to \varphi_2'$

$(\varphi_1 \wedge \varphi_2)' = \mathsf{Product}\ \varphi_1'\ \varphi_2'$

$(\varphi_1 \vee \varphi_2)' = \mathsf{Either}\ \varphi_1'\ \varphi_2'$

Notice: if we introduced $\top$ as a formula, we could define $\top' = \forall \mathsf{C}{:}\star.\mathsf{C} \to \mathsf{C}$

Encoding environments

Remember that **theories** in IPL are **sets**, while **contexts** in λ_ω are **lists**.

Given a finite theory Γ (in IPL), we obtain a context Γ' (in λ_ω) by

- declaring all literals that occur within Γ as available in Γ';
- creating, for each formula $\varphi_i \in \Gamma$, a variable declaration $x_i : \varphi'_i$ in Γ'.

Example: The theory $\{\, X,\ Y \vee Z,\ (X \wedge Y) \Rightarrow \bot \,\}$ (in IPL) can be represented by the following context (in λ_ω):

$$X : \star,\ Y : \star,\ Z : \star,\ x_1 : X,\ x_2 : \mathsf{Either}\ Y\ Z,\ x_3 : (\mathsf{Product}\ X\ Y) \to (\forall C.C)$$

Notice:

- each introduced variable x_i represents an **assumed construction** (a proof) of the associated formula (type).
- the order of type variables and of declarations may change, i.e. the same theory of IPL can be represented by many distinct contexts in λ_ω.

Axiom rule

The axiom rule allows us to assert something that has been assumed in the theory:

$$\frac{\varphi \in \Gamma}{\Gamma \vdash \varphi} \tag{Ax}$$

Notice: from now on we write the type φ' simply as φ (for the sake of legibility).

We assume

$$\Gamma' = \overline{X : \star},\ x_1 : \varphi_1,\ x_2 : \varphi_2,\ \ldots,\ x_n : \varphi_n$$

where $\varphi = \varphi_k$ for some k ($1 \leq k \leq n$).

The derivation below can be obtained by a combination of the rules VAR and VARWEAKTERM.

$$\frac{(x_k : \varphi_k) \in \Gamma'}{\Gamma' \vdash x_k : \varphi_k}$$

Bottom elimination

The bottom elimination rule says that, for all φ, if we have $\bot$ then we have φ.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \qquad (\bot\text{-E})$$

This fact can be represented by the following instantiation of the rule TYPEAPP

$$\frac{\Gamma \vdash \mathsf{x} : \forall \mathsf{U}.\mathsf{U} \qquad \Gamma \vdash \varphi : \star}{\Gamma \vdash \mathsf{botElim}\ \langle\varphi\rangle\ \mathsf{x} : \varphi} \qquad (\text{TYPEAPP})$$

where

$$\mathsf{botElim} = \Lambda\varphi{:}\star.\ \lambda \mathsf{x}{:}(\forall \mathsf{U}.\mathsf{U}).\ \mathsf{x}\ \langle\varphi\rangle$$

Implication introduction and elimination

The introduction and elimination of implication rules, as shown below,

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \quad (\Rightarrow\text{-I}) \qquad \frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \quad (\Rightarrow\text{-E})$$

are **directly** represented by the type system rules LAM and APP:

$$\frac{\Gamma, x : \varphi \vdash e : \psi}{\Gamma \vdash \lambda x : \varphi . e : \varphi \rightarrow \psi} \quad (\text{LAM}) \qquad \frac{\Gamma \vdash e_1 : \varphi \rightarrow \psi \qquad \Gamma \vdash e_2 : \varphi}{\Gamma \vdash e_1\ e_2 : \psi} \quad (\text{APP})$$

And introduction and elimination

The following rules related to conjunctions

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \land \psi}\ (\land\text{-I}) \qquad \frac{\Gamma \vdash \varphi \land \psi}{\Gamma \vdash \varphi}\ (\land\text{-E1}) \qquad \frac{\Gamma \vdash \varphi \land \psi}{\Gamma \vdash \psi}\ (\land\text{-E2})$$

are represented by the following type derivations

$$\frac{\Gamma \vdash e_1 : \varphi \qquad \Gamma \vdash e_2 : \psi}{\Gamma \vdash \mathsf{pair}\ \langle\varphi\rangle\ \langle\psi\rangle\ e_1\ e_2 : \mathsf{Product}\ \varphi\ \psi}$$

$$\frac{\Gamma \vdash e : \mathsf{Product}\ \varphi\ \psi}{\Gamma \vdash \mathsf{fst}\ \langle\varphi\rangle\ \langle\psi\rangle\ e : \varphi} \qquad \frac{\Gamma \vdash e : \mathsf{Product}\ \varphi\ \psi}{\Gamma \vdash \mathsf{snd}\ \langle\varphi\rangle\ \langle\psi\rangle\ e : \psi}$$

Or introduction and elimination

The rules related to disjunctions

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \lor \psi} \quad (\lor\text{-I1})$$

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \lor \psi} \quad (\lor\text{-I2})$$

$$\frac{\begin{array}{c}\Gamma \vdash \varphi \lor \psi \\ \Gamma \vdash \varphi \Rightarrow \xi \\ \Gamma \vdash \psi \Rightarrow \xi\end{array}}{\Gamma \vdash \xi} \quad (\lor\text{-E})$$

are represented by the following type derivations

$$\frac{\Gamma \vdash \mathsf{a} : \varphi \qquad \Gamma \vdash \psi : \star}{\Gamma \vdash \mathsf{left}\ \langle\varphi\rangle\ \langle\psi\rangle\ \mathsf{a} : \mathsf{Either}\ \varphi\ \psi}$$

$$\frac{\Gamma \vdash \mathsf{b} : \psi \qquad \Gamma \vdash \varphi : \star}{\Gamma \vdash \mathsf{right}\ \langle\varphi\rangle\ \langle\psi\rangle\ \mathsf{b} : \mathsf{Either}\ \varphi\ \psi}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{Either}\ \varphi\ \psi \\ \Gamma \vdash \mathsf{f} : \varphi \to \xi \\ \Gamma \vdash \mathsf{g} : \psi \to \xi\end{array}}{\Gamma \vdash \mathsf{case}\ \langle\varphi\rangle\ \langle\psi\rangle\ \langle\xi\rangle\ \mathsf{e}\ \mathsf{f}\ \mathsf{g} : \xi}$$

Deductions of IPL in λ_ω

Since all the rules of IPL can be encoded in λ_ω by a *set of type derivations*, any **combination** of such derivations gives us valid proofs in IPL.

We can use this sets of rules in λ_ω as a **proof assistant** for IPL.

Given a theory Γ and formula φ in IPL, we say that $\mathsf{e} \in \Lambda_\omega$ is a **proof-term** of φ iff $\Gamma'' \vdash \mathsf{e} : \varphi'$ where

1. e is constructed as a combination of the following forms
   - x (axiom)
   - $\mathsf{botElim}\ \langle \varphi_1 \rangle\ \mathsf{e}_1$ (bottom)
   - $\lambda \mathsf{x} : \varphi_1 . \mathsf{e}_1, (\mathsf{e}_1\ \mathsf{e}_2)$ (implication)
   - $\mathsf{left}\ \langle \varphi_1 \rangle\ \langle \varphi_2 \rangle\ \mathsf{e}_1, \mathsf{right}\ \langle \varphi_1 \rangle\ \langle \varphi_2 \rangle\ \mathsf{e}_1, \mathsf{case}\ \langle \varphi_1 \rangle\ \langle \varphi_2 \rangle\ \langle \varphi_3 \rangle\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3$ (disjunction)
   - $\mathsf{pair}\ \langle \varphi_1 \rangle\ \langle \varphi_2 \rangle\ \mathsf{e}_1\ \mathsf{e}_2, \mathsf{fst}\ \langle \varphi_1 \rangle\ \langle \varphi_2 \rangle\ \mathsf{e}_1, \mathsf{snd}\ \langle \varphi_1 \rangle\ \langle \varphi_2 \rangle\ \mathsf{e}_1$ (conjunction)

   and
2. $\Gamma'' = \Gamma'$ + declarations for type variables occuring in e and φ'.

Deductions of IPL in $\lambda\omega$: example

To prove the following theorem of IPL

$$\vdash (A \wedge B) \Rightarrow C \Rightarrow ((B \wedge C) \vee \bot) \qquad = \varphi$$

we can construct the folowing proof-term in λ_ω

$$\begin{aligned} e \quad = \quad & \lambda p{:}\mathsf{Product}\ A\ B.\ \lambda c{:}C. \\ & \mathsf{left}\ \langle \mathsf{Product}\ B\ C\rangle\ \langle \forall U.U\rangle\ (\mathsf{pair}\ \langle B\rangle\ \langle C\rangle\ (\mathsf{snd}\ \langle A\rangle\ \langle B\rangle\ p)\ c) \end{aligned}$$

and verify that $A{:}\star,\ B{:}\star,\ C{:}\star \vdash\ e\ :\ \varphi'$

Exercise: construct proof-terms for the following tautologies in IPL using λ_ω. Remember that $\neg A \equiv A \Rightarrow \bot$.

- $(A \Rightarrow B) \Rightarrow \neg B \Rightarrow \neg A$
- $(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$
- $\neg(A \vee B) \Rightarrow (\neg A \wedge \neg B)$

Classical propositional logic (review)

A complete and consistent deductive system for **classical propositional logic** (CPL) can be obtained by extending the deduction rules of IPL with one of the following: *law of the excluded middle* or *elimination of double negation.*

$$\frac{}{\Gamma \vdash \varphi \vee \neg\varphi}\ (\textsc{lem}) \qquad\qquad \frac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}\ (\neg\neg\textsc{e})$$

We need only one of them, since it is possible to derive both

$$\forall\varphi.(\varphi \vee \neg\varphi) \vdash \forall\psi.(\neg\neg\psi \Rightarrow \psi)$$

and

$$\forall\varphi.(\neg\neg\varphi \Rightarrow \varphi) \vdash \forall\psi.(\psi \vee \neg\psi)$$

Exercise: show the derivations above.

Encoding CPL in λ_ω

It is not possible to build **closed terms** in λ_ω of the following types:

- $\mathsf{lem} : \forall \mathsf{A}{:}\star.\mathsf{Either}\ \mathsf{A}\ (\mathsf{A} \to (\forall \mathsf{U}.\mathsf{U}))$
- $\mathsf{dnegElim} : \forall \mathsf{A}{:}\star.((\mathsf{A} \to (\forall \mathsf{U}.\mathsf{U})) \to (\forall \mathsf{U}.\mathsf{U})) \to \mathsf{A}$

However, if we assume the existence of one of them in the context (for example, lem):

- the logic is **not** trivialized (not everything is provable), and
- the formulas that can be proven (using only the previously encoded rules + lem) correspond exactly to the theorems of the *classical propositional logic*.

This approach is actually used in proof assistants such as Coq: a classical proof of φ can be represented by a constructive proof of

$$(\forall \mathsf{A}{:}\star.\mathsf{A} \vee \neg \mathsf{A}) \to \varphi$$

Deductions of CPL in λ_ω

Let us consider the following deductions of CPL:

- $\varphi \Rightarrow \psi \vdash \neg\psi \Rightarrow \neg\varphi$
- $\neg\varphi \Rightarrow \neg\psi \vdash \psi \Rightarrow \varphi$

The first deduction is constructive, therefore it is also a deduction of IPL.

The second deduction is not constructive, and therefore is a deduction of CPL only, requiring either LEM or $\neg\neg$E.

We will now develop both deductions, using the natural deduction rules and their respective encodings in λ_ω.

Syntax for abstractions in λ_ω

Let us try to simplify the notational aspects of λ_ω without altering it.

In λ_ω we have three abstractions, each with its respective functional type:

Abstraction	Functional "type"	Condition
$\lambda x : \tau_1 . e_2$	$\tau_1 \to \tau_2$	assuming $e_2 : \tau_2$
$\Lambda X : \kappa_1 . e_2$	$\forall X : \kappa_1 . \tau_2$	assuming $e_2 : \tau_2$
$\lambda X : \kappa_1 . \tau_2$	$\kappa_1 \to \kappa_2$	assuming $\tau_2 : \kappa_2$

Notice that the first and third functional "types" ignore (at the type level) the name of the argument, but the second one needs it for binding purposes.

It is possible to represent all of them using a **unified** syntax. We now introduce the notion of Π-types (pi types) as a generalization of all functional "types" above.

Π-types in λ_ω

We can reuse the same syntax for each abstraction (lowercase lambda) and its functional “type” (Π-type):

Abstraction	Functional “type”	Condition
$\lambda x : \tau_1 . e_2$	$\Pi x : \tau_1 . \tau_2$	assuming $e_2 : \tau_2$
$\lambda X : \kappa_1 . e_2$	$\Pi X : \kappa_1 . \tau_2$	assuming $e_2 : \tau_2$
$\lambda X : \kappa_1 . \tau_2$	$\Pi X : \kappa_1 . \kappa_2$	assuming $\tau_2 : \kappa_2$

We can say that the “old” arrow types are just Π-types in which the name of the variable/type variable is irrelevant/not used:

$$\begin{aligned} \tau_1 \to \tau_2 &\quad\equiv\quad \Pi\_ : \tau_1 . \tau_2 \\ \kappa_1 \to \kappa_2 &\quad\equiv\quad \Pi\_ : \kappa_1 . \kappa_2 \end{aligned}$$

Notice: Π-types were actually introduced to represent **dependent types**. The *syntactical generalization* above is just a nice side-effect.

Simplified syntax for λ_ω

It is possible to collapse terms, types, kinds and superkind into the same syntactic category, and give the task of distinguishing them to type judgements.

We may also use only one set of variables for both terms and types.

$$
\begin{array}{lcll}
\mathsf{x} & \in & \mathsf{Var} & \text{(variables)} \\
\\
\mathcal{E} & \in & \mathsf{Exp} & \text{(expressions)} \\
\mathcal{E} & ::= & \mathsf{x} \mid \Box \mid \star \mid (\mathcal{E}_1\ \mathcal{E}_2) \mid \lambda \mathsf{x}\!:\!\mathcal{E}_1.\ \mathcal{E}_2 \mid \Pi \mathsf{x}\!:\!\mathcal{E}_1.\ \mathcal{E}_2 & \\
\\
\mathsf{s} & \in & \{\Box, \star\} & \text{(sorts)} \\
\\
\Gamma & \in & \mathsf{Env} & \text{(environments)} \\
\Gamma & ::= & \bullet \mid \Gamma, \mathsf{x}\!:\!\mathcal{E} &
\end{array}
$$

We use uppercase, cursive letters to range over expressions: $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{E}$

Substitution

$$\begin{array}{lcl}
[\_/\_]\_ & : & \mathsf{Exp} \times \mathsf{Var} \times \mathsf{Exp} \to \mathsf{Exp} \\
[\mathcal{A}/\mathsf{y}]\mathsf{x} & = & \begin{cases} \mathcal{A} & (\mathsf{x} = \mathsf{y}) \\ \mathsf{x} & (\mathsf{x} \neq \mathsf{y}) \end{cases} \\
[\mathcal{A}/\mathsf{y}]\Box & = & \Box \\
[\mathcal{A}/\mathsf{y}]\star & = & \star \\
[\mathcal{A}/\mathsf{y}](\mathcal{E}_1\ \mathcal{E}_2) & = & [\mathcal{A}/\mathsf{y}]\mathcal{E}_1\ [\mathcal{A}/\mathsf{y}]\mathcal{E}_2 \\
[\mathcal{A}/\mathsf{y}](\lambda \mathsf{x}{:}\mathcal{E}_1.\ \mathcal{E}_2) & = & \lambda \mathsf{x}{:}[\mathcal{A}/\mathsf{y}]\mathcal{E}_1.\ [\mathcal{A}/\mathsf{y}]\mathcal{E}_2 \\
[\mathcal{A}/\mathsf{y}](\Pi \mathsf{x}{:}\mathcal{E}_1.\ \mathcal{E}_2) & = & \Pi \mathsf{x}{:}[\mathcal{A}/\mathsf{y}]\mathcal{E}_1.\ [\mathcal{A}/\mathsf{y}]\mathcal{E}_2
\end{array}$$

We assume Barendregt's convention when propagating the substitution towards $\mathcal{E}_2$ in lambda's and pi's.

Type system

The type system is the smallest relation $(\_ \vdash \_ : \_) \subseteq \mathsf{Env} \times \mathsf{Exp} \times \mathsf{Exp}$ satisfying:

$$\frac{}{\bullet \vdash \star : \Box}\ (\textsc{sort})$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \mathcal{A} : s}{\Gamma, x{:}\mathcal{A} \vdash x : \mathcal{A}}\ (\textsc{var})$$

$$\frac{\Gamma \vdash \mathcal{A} : \mathcal{B} \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \mathcal{C} : s}{\Gamma, x{:}\mathcal{C} \vdash \mathcal{A} : \mathcal{B}}\ (\textsc{weak})$$

$$\frac{\Gamma \vdash \mathcal{A} : s_1 \qquad \Gamma, x{:}\mathcal{A} \vdash \mathcal{B} : s_2}{\Gamma \vdash \Pi x{:}\mathcal{A}.\mathcal{B} : s_2}\ (\textsc{form})$$

$$\frac{\Gamma \vdash \mathcal{E}_1 : \Pi x{:}\mathcal{A}.\mathcal{B} \qquad \Gamma \vdash \mathcal{E}_2 : \mathcal{A}}{\Gamma \vdash \mathcal{E}_1\ \mathcal{E}_2 : [\mathcal{E}_2/x]\mathcal{B}}\ (\textsc{app})$$

$$\frac{\Gamma, x{:}\mathcal{A} \vdash \mathcal{E} : \mathcal{B} \qquad \Gamma \vdash \Pi x{:}\mathcal{A}.\mathcal{B} : s}{\Gamma \vdash \lambda x{:}\mathcal{A}.\mathcal{E} : \Pi x{:}\mathcal{A}.\mathcal{B}}\ (\textsc{abs})$$

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

where the allowed combinations of (s_1, s_2) in rule FORM are $\overbrace{(\star, \star)}^{\tau_1 \to \tau_2}$, $\overbrace{(\Box, \Box)}^{\kappa_1 \to \kappa_2}$, $\overbrace{(\Box, \star)}^{\forall X{:}\kappa_1.\tau_2}$

Semantics

Beta-reduction is the smallest relation $(\_ \rightarrow_\beta \_) \subseteq \mathsf{Exp} \times \mathsf{Exp}$ satisfying:

$$\frac{}{(\lambda x{:}\mathcal{A}.\mathcal{E}_1)\ \mathcal{E}_2 \rightarrow_\beta [\mathcal{E}_2/x]\mathcal{E}_1} \quad (\beta)$$

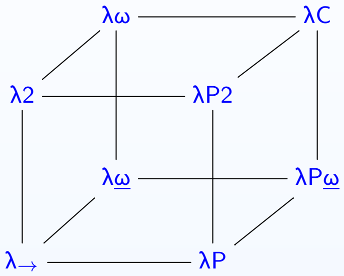
$$\frac{\mathcal{E}_1 \rightarrow_\beta \mathcal{E}_1'}{\mathcal{E}_1\ \mathcal{E}_2 \rightarrow_\beta \mathcal{E}_1'\ \mathcal{E}_2} \quad (\textsc{cpApp1}) \qquad \frac{\mathcal{E} \rightarrow_\beta \mathcal{E}'}{\lambda x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \lambda x{:}\mathcal{A}.\mathcal{E}'} \quad (\textsc{cpAbs2})$$

$$\frac{\mathcal{E}_2 \rightarrow_\beta \mathcal{E}_2'}{\mathcal{E}_1\ \mathcal{E}_2 \rightarrow_\beta \mathcal{E}_1\ \mathcal{E}_2'} \quad (\textsc{cpApp2}) \qquad \frac{\mathcal{A} \rightarrow_\beta \mathcal{A}'}{\Pi x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \Pi x{:}\mathcal{A}'.\mathcal{E}} \quad (\textsc{cpPi1})$$

$$\frac{\mathcal{A} \rightarrow_\beta \mathcal{A}'}{\lambda x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \lambda x{:}\mathcal{A}'.\mathcal{E}} \quad (\textsc{cpAbs1}) \qquad \frac{\mathcal{E} \rightarrow_\beta \mathcal{E}'}{\Pi x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \Pi x{:}\mathcal{A}.\mathcal{E}'} \quad (\textsc{cpPi2})$$

Both $\twoheadrightarrow_\beta$ and $=_\beta$ are defined as usual.

Lambda cube



Calculus	(s_1, s_2) **allowed in rule FORM**			
$\lambda_{\rightarrow}$	$(\star, \star)$			
$\lambda 2$	$(\star, \star)$	$(\square, \star)$		
$\lambda\underline{\omega}$	$(\star, \star)$		$(\square, \square)$	
λP	$(\star, \star)$			$(\star, \square)$
$\lambda\omega$	$(\star, \star)$	$(\square, \star)$	$(\square, \square)$	
$\lambda P2$	$(\star, \star)$	$(\square, \star)$		$(\star, \square)$
$\lambda P\underline{\omega}$	$(\star, \star)$		$(\square, \square)$	$(\star, \square)$
λC	$(\star, \star)$	$(\square, \star)$	$(\square, \square)$	$(\star, \square)$

Some history regarding the lambda cube

1960s-1980s: Introduction of dependent types (Π-types) as a generalization of products and universal quantification (de Bruijn, Martin-Löf, Howard).
https://en.wikipedia.org/wiki/Dependent_type

1960s-1970s: Automath (de Bruijn) (pronounced as "de Brown")
https://en.wikipedia.org/wiki/Automath

1970s-1980s: Intuitionistic Type Theories (Martin-Löf)
https://en.wikipedia.org/wiki/Intuitionistic_type_theory

1980s-2000s: Calculus of Constructions/Coq (Coquand, Huet)
https://en.wikipedia.org/wiki/Calculus_of_constructions

1980s-2000s: Logic Frameworks (Harper et al.), Twelf (Pfenning et al.)
https://en.wikipedia.org/wiki/Twelf

1990s: Lambda Cube, Pure Type Systems (Barendregt et al.)
https://en.wikipedia.org/wiki/Lambda_cube
https://en.wikipedia.org/wiki/Pure_type_system

Content

Limitations of Propositional Logic

In propositional logic the basic unit of information is the *atomic symbol*: a complete and indivisible idea that can be *true* or *false*.

Consider the following sentences

- All *birds* fly $= \mathsf{q}$
- **Penguins** are *birds* $= \mathsf{p}$
- **Penguins** fly $= \mathsf{r}$

As each sentence represents a different idea, we would represent it using different symbols.

However, ideas have *common elements*, and this information is not captured in propositional symbols.

We don't have $\mathsf{p}, \mathsf{q} \models \mathsf{r}$ in propositional logic, although intuitively, r is a consequence of p and q.

Predicate Logic

Predicate logic allows to talk about *objects/individuals* of some domain, *properties* of these individuals, and relations between them.

A = Brasília is the capital of Brasil

Components:

- An object: the city Brasília
- An object property: being the capital of Brazil
- Formalized as: `CapitalOfBrasil(brasília)`

Components (alternative decomposition)

A = Brasília is the capital of Brasil

- Two objects: the city Brasília and the country Brazil
- A relation (property) of the pair of objects: being the capital of a country
- Formalized as: `CapitalOf(brasília, brasil)`

Representing objects

The construction of terms is based on three sets of symbols:

- set **Con** of **constant symbols** $\mathsf{c}, \mathsf{c}_1, \mathsf{c}_2, \ldots$
- set **Fun** of **functional symbols** (each with its arity) $\mathsf{f}, \mathsf{g}, \mathsf{h}, \ldots$
- set **Var** of **variables** (countable and infinite) $\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots$

The set **Term** of terms is defined by the following abstract grammar:

$$\mathsf{t} \quad ::= \quad \mathsf{c} \mid \mathsf{x} \mid \mathsf{f}\,(\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}) \qquad \text{(arity of f is n)}$$

Representing properties and relations

We assume a set **Pred** of predicate symbols P, Q, $\dots$ with their arities.

Predicate symbols represent *properties* of objects and *relations* between them.

Examples: `Even`, `Odd`, `CapitalOfBrasil`, `CapitalOf`.

Observations:

- The equality $=$ is a predicate symbol of arity 2 that will deserve a special treatment later
- The symbols $\bot$ and $\top$ are also special predicate symbols of arity 0

Formulas

The set $\mathcal{F}$ of formulas of first order predicate logic is defined by the abstract grammar:

$$\begin{array}{rcll}
\varphi & ::= & \bot \mid \top \mid t_1 = t_2 \mid P\,(t_1, \ldots, t_n) & \textit{atomic formulas} \\
& \mid & \varphi \land \psi \mid \varphi \lor \psi \mid \varphi \Rightarrow \psi & \textit{formulas with propositional structure} \\
& \mid & \forall x.\varphi \mid \exists x.\varphi & \textit{quantified formulas}
\end{array}$$

As before we write:

- $\neg\,\varphi$ for $\varphi \Rightarrow \bot$
- $\varphi \Leftrightarrow \psi$ for $(\varphi \Rightarrow \psi) \land (\psi \Rightarrow \varphi)$

Formulas: examples

In the following example of formulas we assume that the domain of discourse is the set of natural numbers and that:

- $0, 1 \in \mathbf{Con}$
- $\mathsf{succ}, \mathsf{sum} \in \mathbf{Fun}$
- $\mathsf{Even}, \mathsf{Odd}, \mathsf{Greater} \in \mathbf{Pred}$

Examples of formulas:

$0 = 1$
$\mathtt{Even}(x)$
$\mathtt{Even}(1) \wedge \mathtt{Odd}(0)$
$\mathtt{Odd}(\mathsf{succ}(0))$
$\mathtt{Odd}(\mathsf{succ}(x))$
$\forall x.\mathtt{Even}(x)$
$\exists x.\mathtt{Even}(x)$
$\forall x.(\mathtt{Even}(x) \Rightarrow \mathtt{Odd}(\mathsf{succ}(x)))$
$\forall x.((x = 0) \vee \mathtt{Greater}(\mathsf{sum}(x, x), x))$
$\neg(x = y) \Rightarrow \neg(\mathsf{succ}(x) = \mathsf{succ}(y))$

Precedence

Quantifiers can be seem as unary operators and have the same precedence as negation.

1º	$\neg$, $\forall$, $\exists$
2º	$\wedge$
3º	$\vee$
4º	$\rightarrow$

Example:

$$\forall x.P(x,x) \Rightarrow \neg\exists y.P(x,y) \wedge \forall z.Q(z)$$

=

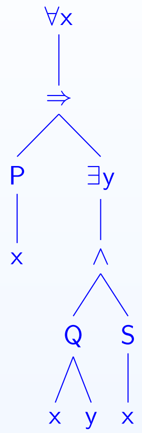
$$(\forall x.P(x,x)) \Rightarrow ((\neg(\exists y.P(x,y))) \wedge (\forall z.Q(z)))$$

Syntax tree of a formula

Formula

$\forall x. (P(x) \Rightarrow \exists y. (Q(x,y) \land S(x)))$

Corresponding syntax tree $\Rightarrow$



Free and bound variables

A formula can have none, one, or several variables.

Depending on where it occurs in a formula, each **occurrence** of a variable can be *bound* or *free*.

Occurrence of variable x in a formula φ:

- *Bound:* if it is under an $\exists x$ or $\forall x$ in the syntax tree of φ
- *Free:* otherwise

Names for bound variables

Two formulas that differ only on the names of their bound variables are equivalent

$$\forall x.(Q(y) \land \exists x.P(x)) \quad \equiv \quad \forall x(Q(y) \land \exists y.P(y))$$

Important: to change the name of a variable bound by a quantifier we have to change the name of all the occurrences bound by that same quantifier.

Important: we should also make sure that no free variable will be captured.

$$\forall x(Q(y) \land \exists x P(x)) \quad \equiv \quad \forall z(Q(y) \land \exists x P(x))$$

ok: no occurrence of x bound to the external $\forall x$

$$\forall x(Q(y) \land \exists x P(x)) \quad \not\equiv \quad \forall y(Q(y) \land \exists x P(x))$$

problem: free variable y has been captured

Substitution

$[\_/\_]\_ : \text{Term} \times \text{Var} \times \mathcal{F} \longrightarrow \mathcal{F}$

Notation: $[t/x]\varphi$

Semantic: replace in φ every *free ocorrence* of x by the term t.

Condition: at the position where x occur free in φ, no free variable of t can have the same name of variable bound by a quantifier.

Examples:

$[c/y]\ \forall x.P(x,y)$ returns $\forall x.P(x,c)$ OK!
$[x/y]\ \forall x.P(x,y)$ Wrong! Free variable is captured!
$[x/y]\ \forall z.P(z,y)$ returns $\forall z.P(z,x)$ OK!
$[c/y]\ \ (\exists y.Q(y) \wedge P(x,y))$ returns $\exists y.Q(y) \wedge P(x,c)$ OK!

It is always possible to rename bound variables to avoid capture of free variables.

Natural Deduction for Intuitionistic FOL

We keep the natural deduction rules of IPL for formulas of predicate logic with propositional structure

We add new rules to deal with quantifiers and equality

Natural deduction in the "traditional style"

- $\Gamma \vdash \varphi$ if we can construct a proof of φ from assumptions in Γ.
- proofs are trees where
  - φ is the root ,
  - leaves are (discharged) hypotheses ou assumptions in Γ,
  - branches are formed by the application of introduction rules and elimination rules

Rule $\forall$ elimination

If we have $\forall x.\varphi$ then it is possible to conclude the formula φ replacing the variable x by any term:

$$\frac{\forall x.\varphi}{[t/x]\,\varphi} \qquad (\forall_E)$$

Example:

$$\frac{\forall x.\,(M(x) \lor H(x))}{M(c) \lor H(c)}\,[\forall_E]$$

The formula obtained is the result of $[c/x]\,(M(x) \lor H(x))$

Rule $\forall$ introduction

$$\frac{\varphi_x}{\forall x.\varphi} \qquad (\forall_I)$$

If the variable x does not appear free in any hypothesis on which φ_x depends. i.e. in any undischarged hypothesis in the derivation of φ_x.

Example: proof that $\forall x.(P(x) \Rightarrow Q(x)), \forall x.P(x) \vdash \forall x.Q(x)$.

$$\cfrac{\cfrac{\cfrac{\forall x.(P(x) \Rightarrow Q(x))}{P(x) \Rightarrow Q(x)}\,[\forall_E] \qquad \cfrac{\forall x.P(x)}{P(x)}\,[\forall_E]}{Q(x)}\,[\rightarrow_E]}{\forall x.Q(x)}\,[\forall_I]$$

Example of wrong use of $\forall$ introduction

Proof (with an error) that $\forall x.(P(x) \Rightarrow Q(x)), P(x) \vdash \forall x.Q(x)$.

$$\dfrac{\dfrac{\dfrac{\forall x.(P(x) \Rightarrow Q(x))}{P(x) \Rightarrow Q(x)}\,[\forall_E] \qquad P(x)}{Q(x)}\,[\to_E]}{\forall x.Q(x) \quad \text{Wrong !!!!}}\,[\forall_I]$$

When we obtain $Q(x)$ in the proof, the x in $Q(x)$ is not **any** x. It is an x that also has to satisfy property P. Hence, we cannot conclude that all elements have property Q.

Rule for $\exists$ introduction

This rule is quite intuitive: if we have a formula valid for some particular term, we can conclude that there is some x for which the formula is valid

$$\frac{\varphi_t}{\exists x.\varphi} \qquad (\exists_I)$$

Example:

$$\frac{M(c)}{\exists x.\, M(x)}\ [\exists_I]$$

Rule $\exists$ elimination

A $\exists x.\varphi$ indicates the existence of an object with property φ, but it does not specify which object is that.

So from formula $\exists x.\varphi$ we can conclude any formula ψ that we are able to derive from the formula φ_x but the x should not appear free in other hypothesis for the proof of ψ and should not appear free in ψ.

$$\frac{\exists x.\varphi \qquad \begin{array}{c}\cancel{\varphi_x}\\ \vdots \\ \psi\end{array}}{\psi} \quad (\exists_E)$$

If x is not free in ψ or in any other hypothesis of the subderivation of ψ other that φ_x

Example

Sequent: $\exists x.P(x), \ \forall y.(P(y)TOQ(y)) \vdash \exists x.Q(x)$

$$\dfrac{\exists x.P(x) \qquad \dfrac{\dfrac{\cancel{P(x)} \qquad \dfrac{\forall y.(P(y) \Rightarrow Q(y))}{P(x) \rightarrow Q(x)}\,[\forall_E]}{Q(x)}\,[\rightarrow_E]}{\exists x.Q(x)}\,[\exists_I]}{\exists x.Q(x)}\,[\exists_E]$$

Rule $\exists$ elimination - alternative

Since $\exists x.\varphi$ indicates the existence of an object with property φ, but it does not specify which object is that, can only conclude from $\exists x.\varphi$ a formula ψ if ψ can be proved from for any object such that φ_x

Hence, the rule $\exists_E$ can also be written as:

$$\frac{\exists x.\varphi \qquad (\forall x.\varphi) \Rightarrow \psi}{\psi} \qquad (\exists_E)$$

Example - alternative proof

Sequent: $\exists x.P(x),\ \ \forall y.(P(y) \Rightarrow Q(y)) \vdash \exists x.Q(x)$

$$
\dfrac{\exists x.P(x) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\cancel{\forall x.P(x)}^{\,1}}{P(x)}[\exists_I] \qquad \dfrac{\forall y.(P(y) \Rightarrow Q(y))}{P(x) \Rightarrow Q(x)}[\forall_E]}{Q(x)}[\rightarrow_E]}{\exists x.Q(x)}[\exists_I]}{(\forall x.P(x)) \Rightarrow \exists x.Q(x)}[\Rightarrow_I, 1]}{\exists x.Q(x)}[\exists_E]
$$

Rules for Identity

The rule for introduction of identify states that every term is equal to itself

$$\frac{}{\mathsf{t} = \mathsf{t}} \qquad (=_{\mathsf{I}})$$

The rule of elimination of equality states that it is allowed to replace equals by equals inside other formulas:

$$\frac{\mathsf{t}_1 = \mathsf{t}_2 \qquad [\mathsf{t}_1/\mathsf{x}]\,\varphi}{[\mathsf{t}_2/\mathsf{x}]\,\varphi} \qquad (=_{\mathsf{E}})$$

Exemplo de prova

Sequent: $c = z \ \vdash \ z = c$

$$\dfrac{c = z \qquad \dfrac{}{c = c}\,[=_I]}{z = c}\,[=_E]$$

Note that

$$(c = c) \text{ is the result of } [c/x]\,(x = c)$$
$$(z = c) \text{ is the result of } [z/x]\,(x = c)$$

Note also that the proof above shows that equality is symmetric.

Exercise: Show that equality is transitive.

All rules for natural deduction

Recall that $\neg\varphi \equiv \varphi \to \bot$

Conjunction:

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi}\ (\wedge i)$$

$$\frac{\varphi \wedge \psi}{\varphi}\ (\wedge e_1)$$

$$\frac{\varphi \wedge \psi}{\psi}\ (\wedge e_2)$$

Disjunction:

$$\frac{\varphi}{\varphi \vee \psi}\ (\vee i_1)$$

$$\frac{\psi}{\varphi \vee \psi}\ (\vee i_2)$$

$$\frac{\varphi \vee \psi \quad \varphi \Rightarrow \chi \quad \psi \Rightarrow \chi}{\chi}\ (\vee e\,)$$

Implication:

$$\frac{\begin{array}{c}\cancel{\varphi}\\ \vdots\\ \psi\end{array}}{\substack{\varphi\\ \psi}}\ (\to i)$$

$$\frac{\varphi \quad \varphi \Rightarrow \psi}{\psi}\ (\to e)$$

Negation:

$$\frac{\bot}{\varphi}\ (\bot e)$$

$$\frac{\begin{array}{c}\cancel{\neg\varphi}\\ \vdots\\ \bot\end{array}}{\varphi}\ (\mathrm{RAA})$$

Intuitionistc First Order Predicate Logic does not have rule RAA

All rules of natural deduction

Universal quantifier :

$$\frac{\varphi_x}{\forall x.\varphi} \quad (\forall_I)$$

$(*)$

$$\frac{\forall x.\varphi}{[t/x]\,\varphi} \quad (\forall_E)$$

Existential quantifier :

$$\frac{\varphi_t}{\exists x.\varphi} \quad (\exists_I)$$

$$\frac{\exists x.\varphi \qquad \begin{matrix}\cancel{\varphi_x}\\ \vdots \\ \psi\end{matrix}}{\psi} \quad (\exists_E)$$

$(**)$

Equality:

$$\frac{}{t = t} \quad (=_I)$$

$$\frac{t_1 = t_2 \qquad [t_1/x]\,\varphi}{[t_2/x]\,\varphi} \quad (=_E)$$

$(*)$ x is not free in hypothesis on which φ_x depends

$(**)$ x is not free in ψ, or in hypothesis of the derivation of ψ, other than φ_x

Sequent style natural deduction

All the rules we have seem for propositional logic plus:

$$\frac{\Gamma \vdash \varphi_x}{\Gamma \vdash \forall x.\varphi} \qquad (\forall_i)$$

x is not free in Γ

$$\frac{\Gamma \vdash \forall x.\varphi}{\Gamma \vdash [t/x]\ \varphi} \qquad (\forall_e)$$

$$\frac{\Gamma \vdash \varphi_t}{\Gamma \vdash \exists x.\varphi} \qquad (\exists_i)$$

$$\frac{\Gamma \vdash \exists x.\varphi \qquad \Gamma, \varphi \vdash \psi}{\Gamma \vdash \psi} \qquad (\exists_e)$$

x is not free in ψ, or in Γ

BHK interpretation

BHK interpretation given to propositional logic extends to 1st order predicate logic

Interpretation of $\top$ and $\bot$ as in propositional logic

Interpretation of atomic formulas $P(t_1, \ldots t_n)$ is left unspecified

Interpretation of quantified formulas :

- A proof of $\forall x.\varphi$ is a method transforming every term t into a proof of proposition $[t/x]\varphi$
- A proof of $\exists X.\varphi$ is a pair $< a, b >$ with 1st component an element of the domain of discourse and, as second component, a proof of $[a/x]\varphi$

Exercises

Prove the following sequents in both styles of natural deduction:

1. $\forall x.R(x) \vdash \exists y.R(y)$
2. $P(a), \forall x.(P(x) \Rightarrow Q(a)) \vdash Q(a)$
3. $\exists x.(\neg P(x) \vee Q(x)), \forall x.P(x) \vdash \exists x.Q(x)$
4. $\forall x.(\neg P(x) \wedge Q(x)) \vdash \forall x.(P(x) \Rightarrow Q(x))$

Content

Second Order Propositional Logic

Second order propositional logic is (first-order) propositional logic extended with universal and existential quantification over propositional variables.

We use A, B, C, X, Y ... over propositional variables

The set Prop2 can be defined by the following abstract grammar:

$$\begin{array}{rcl} \varphi & ::= & \mathsf{A} \mid \bot \mid \top \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \\ & \mid & \forall \mathsf{A}.\varphi \mid \exists \mathsf{A}.\varphi \end{array}$$

Second Order Propositional Logic

- Difference with 1st order propositional logic: $\forall$ and $\exists$ over propositional variables.

- 2nd order propositional logic is **more restricted** than 1st-order predicate logic:
  - all predicate symbols have arity 0 (called propositional letters).
  - That justifies *propositional*, and not *predicate*

- 2nd order propositional logic is **more general** than 1st-order predicate logic:
  - quantification over **propositions** is allowed.
  - That justities *second-order*, and not *first-order*

Second Order Propositional Logic

A propositional variable is a variable over propositions, so a variable of order 2

With quantification over propositional variables, we obtain 2nd-order propositional logic

Then it is possible, for example, to write $\forall C.C \Rightarrow C$, $(\forall P.P) \Rightarrow \bot$

Formula $(\forall P.P) \Rightarrow \bot$ expresses that it is impossible that all propositional formula is true (this cannot be expressed with propositional logic)

In 1st-order propositional logic we say that:

- *formula $\varphi \Rightarrow \varphi$ is a tautology for all formulas φ*

In 2nd-order propositional logic:

- formula $\forall X.X \Rightarrow X$ is a tautology ~~for all formulas φ~~

Second Order Propositional Logic

The conventions for free and bound variables are the same as before

So, formulas such as $\forall A.(A \Rightarrow A)$ and $\forall B.(B \Rightarrow B)$, for example, are identified

Substitution $[\psi/X]\varphi$ is defined as follows:

- $[\psi/X]X = \psi$
- $[\psi/X]Y = Y$, if $X \neq Y$
- $[\psi/X] \perp = \perp$
- $[\psi/X](\varphi \Rightarrow \psi) = [\psi/X]\varphi \Rightarrow [\psi/X]\psi$
- $[\psi/X](\varphi \wedge \psi) = [\psi/X]\varphi \wedge [\psi/X]\psi$
- $[\psi/X](\varphi \vee \psi) = [\psi/X]\varphi \vee [\psi/X]\psi$
- $[\psi/X](\forall B.\varphi) = \forall B.[\psi/X]\varphi$
- $[\psi/X](\exists B.\varphi) = \exists B.[\psi/X]\varphi$

Examples:

- $[X \Rightarrow X/X]\ (X \Rightarrow X) = (X \rightarrow X) \rightarrow (X \rightarrow X)$
- $[Z \Rightarrow W/Y]\ \forall X.(X \Rightarrow Y) = \forall X.(X \Rightarrow (Z \rightarrow W))$
- $[\perp/X](Z \wedge X) = Z\ \wedge \perp$

Impredicativity

The meaning of a formula $\forall \mathsf{A}.\varphi$ depends on the meaning of all formulas $[\psi/\mathsf{A}]\varphi$

So in φ, all occurrences of A can be replaced by some formula ψ.

The formula ψ can be simpler than the formula φ, but it can also be φ itself, or a formula that is more complex than φ.

That is, the meaning of $\forall \mathsf{A}.\varphi$ depends on the meaning of formulas that are possibly more complex than φ .

This is called *impredicativity*

Because of impredicativity, proof methods based on induction on the structure of a formula, fail.

Classical Semantics

We define thee semantics of **classical** 2nd-order propositional logic

Let Var be the set of propositional variables, $\mathbb{B} = \{0, 1\}$ the set of boolean values (0 is false, 1 is true)

An *assignment* of boolean values to propositional variables is a function $\mathsf{v} : \mathsf{Var} \to \mathbb{B}$

Notation: $\mathsf{v} \downarrow^{\mathsf{x}}_{\mathsf{b}}$ is the same as $\mathsf{v} : \mathsf{Var} \to \mathbb{B}$, except that X is mapped to b

The *boolean value of a formula* φ, under assignment v, denoted by $[\![\varphi]\!]_{\mathsf{v}}$, is defined as follows:

$$
\begin{array}{rcl}
[\![\bot]\!]_{\mathsf{v}} & = & 0 \\
[\![\mathsf{P}]\!]_{\mathsf{v}} & = & \mathsf{v}(\mathsf{P}) \\
[\![\varphi \Rightarrow \psi]\!]_{\mathsf{v}} & = & \neg_{\mathbb{B}} [\![\varphi]\!]_{\mathsf{v}} \ \ \vee_{\mathbb{B}} \ \ [\![\psi]\!]_{\mathsf{v}} \\
[\![\varphi \wedge \psi]\!]_{\mathsf{v}} & = & [\![\varphi]\!]_{\mathsf{v}} \ \ \wedge_{\mathbb{B}} \ \ [\![\psi]\!]_{\mathsf{v}} \\
[\![\varphi \vee \psi]\!]_{\mathsf{v}} & = & [\![\varphi]\!]_{\mathsf{v}} \ \ \vee \_\mathbb{B} \ \ [\![\psi]\!]_{\mathsf{v}} \\
[\![\forall \mathsf{X}.\varphi]\!]_{\mathsf{v}} & = & [\![\varphi]\!]_{\mathsf{v}\downarrow^{\mathsf{x}}_{0}} \ \ \wedge_{\mathbb{B}} \ \ [\![\varphi]\!]_{\mathsf{v}\downarrow^{\mathsf{x}}_{1}} \\
[\![\exists \mathsf{X}.\varphi]\!]_{\mathsf{v}} & = & [\![\varphi]\!]_{\mathsf{v}\downarrow^{\mathsf{x}}_{0}} \ \ \vee_{\mathbb{B}} \ \ [\![\varphi]\!]_{\mathsf{v}\downarrow^{\mathsf{x}}_{1}}
\end{array}
$$

BHK interpretation

BHK interpretation given to (1st-order) intuitionistic propositional logic extends to 2n-order

Interpretation of atomic propositions is left unspecified (as in propositional logic)

Interpretation as in 1st order case, plus the interpretation of quantified formulas :

- A proof of $\forall X.\varphi$ is a method transforming every proof of any proposition ψ into a proof of proposition $[\psi/X]\varphi$
- A proof of $\exists X.\varphi$ is a pair with 1st component a proposition ψ and, as second component, a proof of proposition $[\psi/X]\varphi$

Intuitionistic 2nd-Order Propositional Logic

The **natural deduction** proof system for second-order propositional logic is an extension of the one for first-order propositional logic.

New rules: introduction and elimination for universal and existential quantification.

These are similar to the rules for quantification in first-order predicate logic (the domain of quantification is different).

The rules will be given in the sequent style only

Natural Deduction for 2nd Order IPL

$$\frac{}{\Gamma, \varphi \vdash \varphi} \quad (\text{Ax})$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \quad (\wedge\text{-E1})$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \quad (\bot\text{-E})$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \quad (\wedge\text{-E2})$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \quad (\rightarrow\text{-I})$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \quad (\vee\text{-I1})$$

$$\frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \quad (\rightarrow\text{-E})$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \psi \vee \varphi} \quad (\vee\text{-I2})$$

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \quad (\wedge\text{-I})$$

$$\frac{\Gamma \vdash \varphi \vee \psi \qquad \Gamma \vdash \varphi \rightarrow \xi \qquad \Gamma \vdash \psi \rightarrow \xi}{\Gamma \vdash \xi} \quad (\vee\text{-E})$$

$$\frac{\Gamma \vdash \forall X.\varphi}{\Gamma \vdash [\psi/X]\varphi} \quad (\forall\text{-E})$$

$$\frac{\Gamma \vdash [\psi/X]\varphi}{\Gamma \vdash \exists X.\varphi} \quad (\exists\text{-I})$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall X.\varphi} \quad (\forall\text{-I})$$

$$\frac{\Gamma \vdash \exists X.\varphi \qquad \Gamma \vdash \forall X.\varphi \rightarrow \psi}{\Gamma \vdash \psi} \quad (\exists\text{-E})$$

($\forall$-i) X not free in Γ

($\exists$-e) X not free in B

Curry-Howard Isomorphism

Here we consider the **minimal** 2nd-order propositional logic (with $\rightarrow$ and $\forall$ only)

Isomorphism between

- formulas of 2nd-order propositional logic and types of $\lambda 2$
- proofs in 2nd-order propositional logic and terms of $\lambda 2$
- normalisation of proofs in 2nd-order propositional logic and evaluation of terms in $\lambda 2$

Proof normalisation is the removal of *proof detours*

Exercises

1. Show that the $\lambda 2$ term $\lambda x : \forall C.C \rightarrow C.\ x\ \langle \forall C.C \rightarrow C \rangle\ x$ corresponds to a proof of the following formula of 2nd order propositional logic $(\forall C.C \Rightarrow C) \Rightarrow (\forall C.C \Rightarrow C)$
2. Show that the $\lambda 2$ term $\lambda x : \forall C.C \rightarrow C.\ x$ corresponds to a proof of the following formula of 2nd order propositional logic $\forall C.C \rightarrow C$

“Order in Logics”

Variables:

- 1st-order variables - over individuals
- 2nd-order variables - over predicates and functions
- 3rd order variables - over predicates and functions (over predicates and functions)

Logics:

- **1st Order** Predicate Logic (FOL) - 1st order variables, quantifiers over these variables
- **2nd Order** Predicate Logic - 2nd order variables, quantifiers over these variables
- **3rd Order** Predicate Logic - 3rd order variables, quantifiers over these variables
- ...
- **High Order** Logic (HOL) - union of all of them

Logics and the Cube

$$\begin{tikzcd}
 & & \text{[I]Prop.L} \arrow[rr, "\text{Quantifiers}"] & & \text{[I]Pred.L} & \\
 & & \lambda\omega \arrow[rr] & & \lambda C \arrow[r, dashed, "\text{Inductive types}"] & \text{CIC} \\
 & \lambda 2 \arrow[ur] \arrow[rr, dashed] & & \lambda P2 \arrow[ur, dashed] & & \\
 & & \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu, dashed] & \\
\lambda \arrow[r, "\text{Simple types}"'] & \lambda_{\rightarrow} \arrow[uu, "\text{Polymorphism}"] \arrow[ur, dashed, "\text{Type constructors}"'] \arrow[rr, dashed, "\text{Dependent types}"'] & & \lambda P \arrow[uu, dashed] \arrow[ur, dashed] & &
\end{tikzcd}$$

at $\lambda_{\rightarrow}$: terms that depend on terms
($\uparrow$): terms that depend on types
($\nearrow$): types that depend on types
($\rightarrow$): types that depend on terms

System	Logics (Intuitionistic)
$\lambda_{\rightarrow}$	(1st order) propositional logic
$\lambda 2$	2nd order propositional logic
$\lambda\underline{\omega}$	Weak higher order propositional logic
$\lambda\omega$	High order propositional logic
λP	(1st order) predicate logic
$\lambda P2$	2nd order predicate logic
$\lambda P\underline{\omega}$	Weak higher order predicate logic
λC	High order predicate logic

Next: Dependent Types

So far we saw:

- $\lambda_{\rightarrow}$ Terms depending on terms (apply terms to terms $\Rightarrow$ obtain a term)
- λ_2 - terms depending on types (apply types to terms $\Rightarrow$ obtain a term)
- λ_ω - types depending on types (apply types to types $\Rightarrow$ obtain a type)

Next:

λC types depending on terms (apply terms to types $\Rightarrow$ obtain a type)

Content

Calculus of constructions

Dependent Types

We’ve already seem the slogan *types* ***depending*** *on types* (in a more precise way: *proper type constructors depending on type constructors*)

But *dependent types* is to be understood as *types* ***depending*** *on* ***terms*** (in a more precise way: *proper type constructors depending on terms*)

With *dependent types*, type inference, performed at compile time, can obtain information otherwise available only at runtime

Example: type checking helps avoid out of bound errors with information about size of arrays/lists expressed in types

Programming Languages with Dependent Types

Polymorphism (terms depending on types) and type constructors (types depending on types) are found in many programming languages.

Programming languages with dependent types, however, are not that common

Programming languages - more oriented to writing programs with some guarantees of correctness

- Agda
- Idris
- F$*$
- Haskell

Proof assistants - more oriented to proof taking advantage of the Curry-Howard Isomorphism

- Coq
- Lean

Motivation (I)

Consider a function `mk_lst_with_zeroes` that takes a natural number n as input and produces a list with n zeros:

$$\begin{array}{lcl}
\texttt{mk\_lst\_with\_zeroes}\ 0 & = & \mathsf{empty} \\
\texttt{mk\_lst\_with\_zeroes}\ 1 & = & \mathsf{cons}\ 0\ \mathsf{empty} \\
\texttt{mk\_lst\_with\_zeroes}\ 2 & = & \mathsf{cons}\ \ 0\ \ (\mathsf{cons}\ \ 0\ \ \mathsf{empty}) \\
\texttt{mk\_lst\_with\_zeroes}\ 3 & = & \mathsf{cons}\ \ 0\ \ (\mathsf{cons}\ \ 0\ \ (\mathsf{cons}\ \ 0\ \ \mathsf{empty})) \\
\dots & &
\end{array}$$

A possible type for `mk_lst_with_zeroes` is $\mathsf{nat} \to \mathsf{natlist}$.

Type checking provides some level of program correctness.

So if a programmer writes $\dots(\texttt{mk\_lst\_with\_zeroes}\ \mathsf{true})\dots$ a type error message is issued at "compiler time".

But if the programmer writes $\dots(\texttt{nth}\ \ (\texttt{mk\_lst\_with\_zeroes}\ 3)\ 5)\dots$ there is nothing the type checking can do.

The expression type checks and an *out of bound error* occurs at runtime.

Motivation (II)

What if the **type** informs that `mk_lst_with_zeroes` 3 is a list of naturals of length 3?

If the type carries that information, by inspecting it, taking the 5th element of `mk_lst_with_zeroes` 3 yields a **type error**

What we need are **types** as follows

- natlist 0 - type of lists of natural numbers of length 0 (empty list)
- natlist 1 - type of lists of natural numbers of length 1
- natlist 2 - type of lists of natural numbers of length 2
- ...

and a type system able to derive **types that depend on terms**:

- `mk_lst_with_zeroes` 0 : `natlist` 0
- `mk_lst_with_zeroes` 1 : `natlist` 1
- `mk_lst_with_zeroes` 2 : `natlist` 2
- ...

Motivation (III)

A more informative type for **reverse**, for example, would express that, if it takes an input of type a natlist of length n then its result is also of type natlist of the same length n.

The dependent type for reverse is:

$$\textbf{reverse\_nat} : \Pi\, n : nat.\ natlist\ n \to natlist\ n$$

This *term to type* type is written with Π, a λ-like variable-binding notation

A type $\Pi x : \tau . \tau'$ takes a term of type τ and produces the type τ' (which can use x)

Here are the dependent types for other functions that operate with lists:

$$\begin{array}{lcl}
\textbf{append\_nat} & : & \Pi\, n : nat. \Pi\, m : nat.\ natlist\ n \to natlist\ m \to natlist\ (n+m) \\
\textbf{cons\_nat} & : & \Pi\, n : nat.\ nat \to natlist\ n \to natlist\ (succ\ n) \\
\textbf{hd\_nat} & : & \Pi\, n : nat.\ natlist\ (succ\ n) \to nat \\
\textbf{tl\_nat} & : & \Pi\, n : nat.\ natlist\ (succ\ n) \to natlist\ n
\end{array}$$

Constructors

Type constructors also play role in dependent types

Observe that **natlist** is a proper type constructor that **depends on a term** !

The type constructor **natlist** takes as input a natural number n (a term) and returns a complete type representing list of natural numbers of length n:

What is the *kind* of constructor **natlist** ?

$$\begin{array}{lclcl}
\mathsf{natlist} & : & \mathsf{nat} \to * & : & \Box \\
 & & & & \\
\mathsf{natlist}\ 0 & : & * & : & \Box \\
\mathsf{natlist}\ 4 & : & * & : & \Box \\
\mathsf{natlist}\ (\mathsf{mult}\ 4\ 5) & : & * & : & \Box \\
\mathsf{natlist}\ (\mathsf{fat}\ 5) & : & * & : & \Box \\
\mathsf{natlist}\ ((\lambda x : \mathsf{nat}.x)\ 3) & : & * & : & \Box
\end{array}$$

Back to the cube: λC - Calculus of Constructions

Simplified syntax:

Abstraction	Functional "type"	Condition
$\lambda x : \tau_1 . e_2$	$\tau_1 \to \tau_2$	assuming $e_2 : \tau_2$
$\Lambda X : \kappa_1 . e_2$	$\forall X : \kappa_1 . \tau_2$	assuming $e_2 : \tau_2$
$\lambda X : \kappa_1 . \tau_2$	$\kappa_1 \to \kappa_2$	assuming $\tau_2 : \kappa_2$
$\Lambda x : \tau_1 . \tau_2$	$\Pi x : \tau_1 . \kappa_2$	assuming $\tau_2 : \kappa_2$

The first and third functional "types" ignore (at the type level) the name of the argument, but the second and fourth need it for binding purposes.

Same syntax for each abstraction (lowercase lambda) and its functional "type" (Π-type):

Abstraction	Functional "type"	Condition
$\lambda x : \tau_1 . e_2$	$\Pi x : \tau_1 . \tau_2$	assuming $e_2 : \tau_2$
$\lambda X : \kappa_1 . e_2$	$\Pi X : \kappa_1 . \tau_2$	assuming $e_2 : \tau_2$
$\lambda X : \kappa_1 . \tau_2$	$\Pi X : \kappa_1 . \kappa_2$	assuming $\tau_2 : \kappa_2$
$\lambda x : \tau_1 . \tau_2$	$\Pi x : \tau_1 . \kappa_2$	assuming $\tau_2 : \kappa_2$

Remember the following for the "old arrow types:

$$\begin{aligned} \tau_1 \to \tau_2 &\equiv \Pi\_ : \tau_1 . \tau_2 \\ \kappa_1 \to \kappa_2 &\equiv \Pi\_ : \kappa_1 . \kappa_2 \end{aligned}$$

Syntax for λC

The simplified syntax for $\lambda\omega$ contemplates λC as well

$$
\begin{array}{lll l}
x & \in & \mathsf{Var} & \text{(variables)} \\
\\
\mathcal{E} & \in & \mathsf{Exp} & \text{(expressions)} \\
\mathcal{E} & ::= & x \mid \Box \mid \star \mid (\mathcal{E}_1\ \mathcal{E}_2) \mid \lambda x : \mathcal{E}_1 .\ \mathcal{E}_2 \mid \Pi x : \mathcal{E}_1 .\ \mathcal{E}_2 & \\
\\
\Gamma & \in & \mathsf{Env} & \text{(environments)} \\
\Gamma & ::= & \bullet \mid \Gamma, x : \mathcal{E} &
\end{array}
$$

Uppercase, cursive letters are used to range over expressions: $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{E}$

Type system

The type system is the smallest relation $(\_ \vdash \_ : \_) \subseteq \mathsf{Env} \times \mathsf{Exp} \times \mathsf{Exp}$ satisfying:

$$\frac{}{\bullet \vdash \star : \Box}\ (\textsc{sort})$$

$$\frac{\Gamma \vdash \mathcal{E}_1 : \Pi x{:}\mathcal{A}.\mathcal{B} \qquad \Gamma \vdash \mathcal{E}_2 : \mathcal{A}}{\Gamma \vdash \mathcal{E}_1\ \mathcal{E}_2 : [\mathcal{E}_2/x]\mathcal{B}}\ (\textsc{app})$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \mathcal{A} : s}{\Gamma, x{:}\mathcal{A} \vdash x : \mathcal{A}}\ (\textsc{var})$$

$$\frac{\Gamma, x{:}\mathcal{A} \vdash \mathcal{E} : \mathcal{B} \qquad \Gamma \vdash \Pi x{:}\mathcal{A}.\mathcal{B} : s}{\Gamma \vdash \lambda x{:}\mathcal{A}.\mathcal{E} : \Pi x{:}\mathcal{A}.\mathcal{B}}\ (\textsc{abs})$$

$$\frac{\Gamma \vdash \mathcal{A} : \mathcal{B} \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \mathcal{C} : s}{\Gamma, x{:}\mathcal{C} \vdash \mathcal{A} : \mathcal{B}}\ (\textsc{weak})$$

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

$$\frac{\Gamma \vdash \mathcal{A} : s_1 \qquad \Gamma, x{:}\mathcal{A} \vdash \mathcal{B} : s_2}{\Gamma \vdash \Pi x{:}\mathcal{A}.\mathcal{B} : s_2}\ (\textsc{form})$$

All combinations of (s_1, s_2) in rule FORM: $(\star, \star)$, $(\Box, \star)$, $(\Box, \Box)$, $(\star, \Box)$

Semantics

Beta-reduction is the smallest relation $(\_ \rightarrow_\beta \_) \subseteq \mathsf{Exp} \times \mathsf{Exp}$ satisfying:

$$\frac{}{(\lambda x{:}\mathcal{A}.\mathcal{E}_1)\ \mathcal{E}_2 \rightarrow_\beta [\mathcal{E}_2/x]\mathcal{E}_1} \quad (\beta)$$

$$\frac{\mathcal{E}_1 \rightarrow_\beta \mathcal{E}_1'}{\mathcal{E}_1\ \mathcal{E}_2 \rightarrow_\beta \mathcal{E}_1'\ \mathcal{E}_2} \quad (\text{cpApp1}) \qquad \frac{\mathcal{E} \rightarrow_\beta \mathcal{E}'}{\lambda x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \lambda x{:}\mathcal{A}.\mathcal{E}'} \quad (\text{cpAbs2})$$

$$\frac{\mathcal{E}_2 \rightarrow_\beta \mathcal{E}_2'}{\mathcal{E}_1\ \mathcal{E}_2 \rightarrow_\beta \mathcal{E}_1\ \mathcal{E}_2'} \quad (\text{cpApp2}) \qquad \frac{\mathcal{A} \rightarrow_\beta \mathcal{A}'}{\Pi x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \Pi x{:}\mathcal{A}'.\mathcal{E}} \quad (\text{cpPi1})$$

$$\frac{\mathcal{A} \rightarrow_\beta \mathcal{A}'}{\lambda x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \lambda x{:}\mathcal{A}'.\mathcal{E}} \quad (\text{cpAbs1}) \qquad \frac{\mathcal{E} \rightarrow_\beta \mathcal{E}'}{\Pi x{:}\mathcal{A}.\mathcal{E} \rightarrow_\beta \Pi x{:}\mathcal{A}.\mathcal{E}'} \quad (\text{cpPi2})$$

Both $\twoheadrightarrow_\beta$ and $=_\beta$ are defined as usual.

Properties of λC

Uniqueness of types. If $\Gamma \vdash \mathcal{A} : \mathcal{B}_1$ and $\Gamma \vdash \mathcal{A} : \mathcal{B}_2$, then $\mathcal{B}_1 =_\beta \mathcal{B}_2$.

Church-Rosser property. If $\mathcal{E} \twoheadrightarrow_\beta \mathcal{E}_1$ and $\mathcal{E} \twoheadrightarrow_\beta \mathcal{E}_2$, then there is $\mathcal{E}_3$ such that $\mathcal{E}_1 \twoheadrightarrow_\beta \mathcal{E}_3$ and $\mathcal{E}_2 \twoheadrightarrow_\beta \mathcal{E}_3$.

Subject Reduction. If $\Gamma \vdash \mathcal{A} : \mathcal{B}$ and $\mathcal{A} \twoheadrightarrow_\beta \mathcal{A}'$, then $\Gamma \vdash \mathcal{A}' : \mathcal{B}$.

Strong Normalization. Every legal expression $\mathcal{E}$ of λC is strongly normalising.

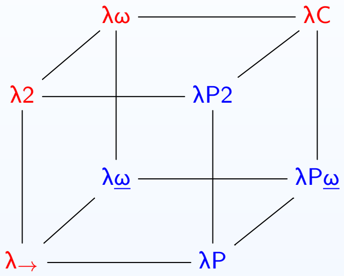
Decidability of Well-typedness. Given a λC expression $\mathcal{A}$, is there a context Γ and an expression $\mathcal{B}$ such that $\Gamma \vdash \mathcal{A} : \mathcal{B}$?

Decidability of Type-checking. Given λC expressions $\mathcal{A}$ and $\mathcal{B}$ and a context Γ. Does $\Gamma \vdash \mathcal{A} : \mathcal{B}$?

Undecidability of Type Inhabitation. Given a λC expression $\mathcal{B}$ and context Γ. Is there λC expression $\mathcal{A}$ such that $\Gamma \vdash \mathcal{A} : \mathcal{B}$?

Calculus of constructions

Adding **dependent types** to $\lambda\omega$ we arrive at λC, the **Calculus of Constructions**



Calculus	(s_1, s_2) **allowed in rule FORM**			
$\lambda\to$	$(\star,\star)$			
$\lambda 2$	$(\star,\star)$	$(\square,\star)$		
$\lambda\underline{\omega}$	$(\star,\star)$		$(\square,\square)$	
λP	$(\star,\star)$			$(\star,\square)$
$\lambda\omega$	$(\star,\star)$	$(\square,\star)$	$(\square,\square)$	
$\lambda P2$	$(\star,\star)$	$(\square,\star)$		$(\star,\square)$
$\lambda P\underline{\omega}$	$(\star,\star)$		$(\square,\square)$	$(\star,\square)$
λC	$(\star,\star)$	$(\square,\star)$	$(\square,\square)$	$(\star,\square)$

Encoding IL in λC

With dependent types it is possible to encode *minimal predicate logic*

Minimal Predicate Logic has only implication and universal quantification

The basic entities of this logic are *propositions*, *sets* and *predicates over sets*

Let's recall the main ideas of *proposition-as-types/proofs-as-terms*:

- if a term b inhabits type B, i.e., if $b : B$, where B is interpreted as a proposition, then we interpret b as a proof of B. We say that b is a proof object
- if there is no inhabitant of the type B (there is no b such that $b : B$), then there is no proof of proposition B, so B must be false

The existence of an inhabitant for a type B (a proof object for proposition B) should be checked in a type system , providing context Γ and term B such that $\Gamma \vdash b : B$

Coding Sets

A set S is coded as a type, so $\mathsf{S} : *$

Elements of S are terms. If a is an element of a set S then $\mathsf{a} : \mathsf{S}$

If S is empty there should be no term a such that $\mathsf{a} : \mathsf{S}$ is derivable

Example: $\mathsf{nat} : *$, $\mathsf{nat} \to \mathsf{nat} : *$, $3 : \mathsf{nat}$, $\lambda \mathsf{n}{:}\mathsf{nat}.\ \mathsf{n} : \mathsf{nat} \to \mathsf{nat}$

Coding Propositions

A proposition A is also coded as a type. So if A is a proposition, then $A : *$

A term p inhabiting such A is a proof object of A

If p is a proof object of A, we write $p : A$

If there is no proof of A then there should be no term p such that $p : A$ is derivable

Coding Predicates

A *predicate* P is a function from a set S to the set of all propositions.

$$\mathsf{P} : \mathsf{S} \to *$$

So, for each $\mathsf{a} : \mathsf{S}$, we have that $\mathsf{P}\ \mathsf{a} : *$

All these $\mathsf{P}\ \mathsf{a}$ are propositions, which are types, so

- if $\mathsf{P}\ \mathsf{a}$ is inhabited, i.e., there is a term t such that $\mathsf{t} : \mathsf{P}\ \mathsf{a}$, then the predicate P holds for a
- if $\mathsf{P}\ \mathsf{a}$ is not inhabited, i.e., there is no term t such that $\mathsf{t} : \mathsf{P}\ \mathsf{a}$, then the predicate P does not hold for a

Coding implication

By Isomorphism of Curry-Howard we identify implication $\mathsf{A} \Rightarrow \mathsf{B}$ with the type $\mathsf{A} \to \mathsf{B}$

By the BHK interpretation, a proof of $\mathsf{A} \Rightarrow \mathsf{B}$ is a mapping from a proof of A to a proof of B

So the truth of implication $\mathsf{A} \Rightarrow \mathsf{B}$ is equivalent to the inhabitation of type $\mathsf{A} \to \mathsf{B}$

Remark: Propositions A and B are "independent" from each other, hence we can write the type as $\mathsf{A} \to \mathsf{B}$ instead of $\Pi \mathsf{x} : \mathsf{A}.\mathsf{B}$

For types that can be written as $\mathsf{A} \to \mathsf{B}$ we can write λC rules for abstraction and application as follows (which are natural deduction rules for implication)

$$\frac{\Gamma \vdash \mathcal{E}_1 : \mathcal{A} \to \mathcal{B} \qquad \Gamma \vdash \mathcal{E}_2 : \mathcal{A}}{\Gamma \vdash \mathcal{E}_1\ \mathcal{E}_2 : \mathcal{B}} \quad (\textsc{app})$$

$$\frac{\Gamma, \mathsf{x} : \mathcal{A} \vdash \mathcal{E} : \mathcal{B} \qquad \Gamma \vdash \mathcal{A} \to \mathcal{B} : \mathsf{s}}{\Gamma \vdash \lambda \mathsf{x} : \mathcal{A}.\mathcal{E} : \mathcal{A} \to \mathcal{B}} \quad (\textsc{abs})$$

Coding Universal Quantification

Now consider the universal quantification $\forall x \in S.\ P(x)$

What is the the interpretation of $\forall x \in S.\ P(x)$ in the Curry-Howard Isomorphism?

$\forall x \in S.\ P(x)$ is true
$\equiv$ for each x in the set S, the proposition $P(x)$ holds
$\equiv$ for each $x \in S$ the type $P\ x$ is inhabited
$\equiv$ there is a function f from each $x \in S$ to an inhabitant of $P\ x$ (f has type $\Pi x{:}S.\ P\ x$)
$\equiv$ there is a function f, with $f : \Pi x{:}S.\ P\ x$
$\equiv$ $\Pi x{:}S.\ P\ x$ is inhabited.

So universal quantification $\forall x \in S.\ P(x)$ is coded as the type $\Pi x{:}S.\ P\ x$

Remarks:

- The logical proposition $P(x)$ (the predicate P for value x) has been coded as the type-theoretic expression $P\ x$
- $\Pi x{:}S.\ P\ x$ is a type depending on a term x (which actually occurs in the body $P\ x$)

Coding Universal Quantification

Rule $\forall E$ is a special case of rule for application in λC.

$$\frac{\Gamma \vdash \forall x \in S.\, P(x) \qquad N \in S}{\Gamma \vdash [N/x]P(x)}\ (\forall\, E) \qquad\qquad \frac{\Gamma \vdash \mathcal{E}_1 : \Pi x{:}\mathcal{A}.\mathcal{B} \qquad \Gamma \vdash \mathcal{E}_2 : \mathcal{A}}{\Gamma \vdash \mathcal{E}_1\ \mathcal{E}_2 : [\mathcal{E}_2/x]\mathcal{B}}\ (\textsc{app})$$

- The $\forall$ in $(\forall E)$ is coded as Π in (APP)
- S corresponds to $\mathcal{A}$
- $P(x)$ in $(\forall E)$ is $\mathcal{B}$ in (APP)
- Γ in $(\forall E)$ have the form $C, D, E, \ldots$ and Γ in (APP) are $x : C, y : D, w : E, \ldots$
- And in (APP) we have *proof objects* for propositions $\Pi x{:}\mathcal{A}.\mathcal{B}$ and $\mathcal{A}$, and a proof object to proposition $[\mathcal{E}_2/x]\mathcal{B}$

Coding Universal Quantification

Rule $\forall\mathsf{I}$ is a special case of rule for abstraction in $\lambda\mathsf{C}$.

$$\frac{\Gamma \vdash \mathsf{P}(\mathsf{x}) \qquad \mathsf{x} \in \mathsf{S} \text{ arbitrary}}{\Gamma \vdash \forall \mathsf{x} \in \mathsf{S}.\, \mathsf{P}(\mathsf{x})} \; (\forall\, \mathrm{I}) \qquad \frac{\Gamma, \mathsf{x} : \mathcal{A} \vdash \mathsf{M} : \mathcal{B} \qquad \Gamma \vdash \Pi \mathsf{x} : \mathcal{A}.\mathcal{B} : \mathsf{s}}{\Gamma \vdash \lambda \mathsf{x} : \mathcal{A}.\, \mathsf{M} : \Pi \mathsf{x} : \mathcal{A}.\mathcal{B}} \; (\textsc{abs})$$

- The 2nd premiss in (ABS) does not occur in $(\forall\mathsf{I})$ so we can dismiss it in our comparison
- Again $\forall$ in $(\forall\mathsf{I})$ is coded as Π in (ABS)
- $\mathsf{P}(\mathsf{x})$ in $(\forall\mathsf{E})$ is $\mathcal{B}$ in (APP)
- Context in $(\forall\mathsf{I})$ have the form $\mathsf{C}, \mathsf{D}, \mathsf{E}, \ldots$ and context in (ABS) have the form $\mathsf{x} : \mathsf{C}, \mathsf{y} : \mathsf{D}, \mathsf{w} : \mathsf{E}, \ldots$
- S has been changed to $\mathcal{A}$
- And, again, (ABS) we have *proof object* M of proposition $\mathcal{B}$, and a proof object $\lambda \mathsf{x} : \mathcal{A}.\, \mathsf{M}$ of proposition $\Pi \mathsf{x} : \mathcal{A}.\mathcal{B}$

Example of a logical derivation in λC

The following is provable in minimal predicate logic:

$$\{\} \vdash \forall x \in S.\ \forall y \in S.\ Q(x, y) \ \Rightarrow\ \forall u \in S.\ Q(u, u)$$

Proof in the *sequent* ND style:

$$\begin{array}{cl}
\hline & [Ax] \\
\forall x \in S.\ \forall y \in S.\ Q(x, y) \vdash \forall x \in S.\ \forall y \in S.\ Q(x, y) & \\
\hline & [\forall_E] \\
\forall x \in S.\ \forall y \in S.\ Q(x, y) \vdash \forall y \in S.\ Q(u, y) & \\
\hline & [\forall_E] \\
\forall x \in S.\ \forall y \in S.\ Q(x, y) \vdash Q(u, u) & \\
\hline & [\forall_I] \\
\forall x \in S.\ \forall y \in S.\ Q(x, y) \vdash \forall u \in S.\ Q(u, u) & \\
\hline & [\rightarrow_I] \\
\varnothing \vdash \forall x \in S.\ \forall y \in S.\ Q(x, y) \ \Rightarrow\ \forall u \in S.\ Q(u, u) &
\end{array}$$

In λC the proposition is

$$\Pi x : S.\ \Pi y : S.\ Q\ x\ y \ \rightarrow\ \Pi u : S.\ Q\ u\ u$$

Example of logical derivation in λC

We have to find an inhabitant for $\Pi x : S.\ \Pi y : S.\ Q\ x\ y\ \rightarrow\ \Pi u : S.\ Q\ u\ u$

We start observing that this is a function type reflecting that the proposition to be proved is an implication.

So, to make the proof shorter we will work with the following "instance”of general rule (ABS) of λC:

$$\frac{\Gamma, z : \mathcal{A} \vdash \mathcal{E} : \mathcal{B}}{\Gamma \vdash \lambda z : \mathcal{A}.\ \mathcal{E} : \mathcal{A} \rightarrow \mathcal{B}} \qquad \text{(ABS)}$$

In our case we have that:

$\mathcal{A}$ is $\Pi x : S.\ \Pi y : S.\ Q\ x\ y$, and

$\mathcal{B}$ is $\Pi u : S.\ Q\ u\ u$

Example of logical derivation in λC

$\Gamma_{SQ} = S : *, Q : S \to S \to *$
$\Gamma_{SQz} = \Gamma_{SQ}, z : \mathcal{A}$
$\Gamma_{SQzu} = \Gamma_{SQ}, z : \mathcal{A}, u : S$
$\mathcal{A} = \Pi x{:}S.\ \Pi y{:}S.\ Q\ x\ y$, and
$\mathcal{B} = \Pi u{:}S.\ Q\ u\ u$

$$\frac{\vdots}{\Gamma_{SQ} \vdash ?? : \mathcal{A} \to B}$$

A proof term for an implication is an abstraction so we know more about $??$:

$$\dfrac{\dfrac{\vdots}{\Gamma_{SQ}, z : \mathcal{A} \vdash ?_{\mathcal{B}} : \Pi u{:}S.\ Q\ u\ u}}{\Gamma_{SQ} \vdash \lambda z : \mathcal{A}.\ ?_{\mathcal{B}} : \mathcal{A} \to B}\ \text{(abs)}$$

A proof term for $\Pi : S.\ Q\ u\ u$ is also an abstraction, so instead of $?_{\mathcal{B}}$ we write:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_{SQz}, u : S \vdash ?_{Quu} : Q\ u\ u}}{\Gamma_{SQ}, z : \mathcal{A} \vdash \lambda u : S.\ ?_{Quu} : \Pi u{:}S.\ Q\ u\ u}\ \text{(abs)}}{\Gamma_{SQ} \vdash \lambda z : \mathcal{A}.\ ?_{\mathcal{B}} : \mathcal{A} \to B}\ \text{(abs)}$$

Example of logical derivation in λC

$\Gamma_{SQ} = S : *, Q : S \to S \to *$
$\Gamma_{SQz} = \Gamma_{SQ}, z : \mathcal{A}$
$\Gamma_{SQzu} = \Gamma_{SQ}, z : \mathcal{A}, u : S$
$\mathcal{A} = \Pi x{:}S.\ \Pi y{:}S.\ Q\ x\ y$, and
$\mathcal{B} = \Pi u{:}S.\ Q\ u\ u$

What would be proof a term $?_{Quu}$ for $Q\ u\ u$? The type of z has all that takes for obtaining $Q\ u\ u$

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_{SQz}\ , u : S \vdash ?_{Quu} : Q\ u\ u}}{\Gamma_{SQ}\ , z : \mathcal{A} \vdash\ \lambda u : S.\ ?_{Quu} : \Pi u{:}S.\ Q\ u\ u}\ (\text{abs})}{\Gamma_{SQ} \vdash\ \lambda z : \mathcal{A}.\ ?_{\mathcal{B}} : \mathcal{A} \to B}\ (\text{abs})$$

The application $z\ u$, according to λC rule (app) has type $[u/x]\mathcal{A}$ which is $\Pi y{:}S.\ Q\ u\ y$

The application $(z\ u)\ u$, according to λC rule (app) has type $[u/y](Q\ u\ y)$ which is $Q\ u\ u$.

So the term $?_{Quu}$ is then $(z\ u)\ u$.

Example of logical derivation in λC

In the proof below:
$\Gamma_{SQ} = S : *, Q : S \to S \to *$
$\Gamma_{SQz} = \Gamma_{SQ}, z : \mathcal{A}$
$\Gamma_{SQzu} = \Gamma_{SQ}, z : \mathcal{A}, u : S$
$\mathcal{A} = \Pi x{:}S.\ \Pi y{:}S.\ Q\ x\ y$, and
$\mathcal{B} = \Pi u{:}S.\ Q\ u\ u$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_{SQzu} \vdash z : \Pi x{:}S.\ \Pi y{:}S.\ Q\ x\ y} \qquad \dfrac{\vdots}{\Gamma_{SQzu} \vdash u : S}}{\Gamma_{SQzu} \vdash z\ u : \Pi y{:}S.\ Q\ u\ y}\,(\text{app}) \qquad \dfrac{\vdots}{\Gamma_{SQzu} \vdash u : S}}{\Gamma_{SQz}, u : S \vdash (z\ u)\ u : Q\ u\ u}\,(\text{app})}{\Gamma_{SQ}, z : \mathcal{A} \vdash \lambda u : S.\ (z\ u)\ u : \Pi u{:}S.\ Q\ u\ u}\,(\text{abs})}{\Gamma_{SQ} \vdash \lambda z : \mathcal{A}.\ \lambda u : S.\ (z\ u)\ u : \mathcal{A} \to B}\,(\text{abs})
$$

We still have to finish proving $\Gamma_{SQzu} \vdash z : \Pi x{:}S.\ \Pi y{:}S.\ Q\ x\ y$ and $\Gamma_{SQzu} \vdash u : S$

Example of logical derivation in λC

In the proof below:
$\Gamma_{SQ} = S : *, Q : S \to S \to *$
$\Gamma_{SQz} = \Gamma_{SQ}, z : \mathcal{A}$
$\Gamma_{SQzu} = \Gamma_{SQ}, z : \mathcal{A}, u : S$

$$\dfrac{\dfrac{\vdots}{\Gamma_{SQ} \vdash S : *}\,(\text{weak}) \qquad z \notin \text{dom}(\Gamma_{SQ}) \qquad \dfrac{\vdots}{\Gamma_{SQ} \vdash A : *}\,(\text{form})}{\dfrac{\Gamma_{SQ}, z : A \vdash S : *}{\Gamma_{SQz}, u : S \vdash u : S}\,(\text{var})}\,(\text{weak})$$

Rules used:

$$\dfrac{}{\bullet \vdash \star : \square} \quad (\textsc{sort}) \qquad \dfrac{\Gamma \vdash \mathcal{A} : \mathcal{B} \qquad x \notin \text{dom}(\Gamma) \qquad \Gamma \vdash \mathcal{C} : s}{\Gamma, x{:}\mathcal{C} \vdash \mathcal{A} : \mathcal{B}} \quad (\textsc{weak})$$

$$\dfrac{x \notin \text{dom}(\Gamma) \qquad \Gamma \vdash \mathcal{A} : s}{\Gamma, x{:}\mathcal{A} \vdash x : \mathcal{A}} \quad (\textsc{var})$$

Encoding *absurdity* λC

Encoding of absurdity:

The λC expression that encodes $\perp$ is the following type:

$$\Pi x : *.\ x$$

Observe that for this expression we have that $(s_1, s_2) = (\Box, *)$ so this type "lives "in $\lambda 2$.

We have seen $\Pi x : *.\ x$ before when we studied $\lambda 2$

There, it was called the empty type (since it has no inhabitants) and we wrote it as

$$\forall C.\ C$$

which, in our more general syntax, is written as $\Pi C : *.\ C$, or simply, $\Pi x : *.\ x$

Encoding negation

Having *absurdity* we also have negation:

$$\neg A \equiv A \to \bot$$

Observe that $A \to \bot$ is an abbreviation for

$$\Pi x : A.\ \ \bot$$

We have both that $A : *$ and $\bot : *$, so here $(s_1, s_2) = (*, *)$. **But** because of the coding of $\bot$ we need at least $\lambda 2$ to code negation.

Conjunction

A conjunction $\mathsf{A} \land \mathsf{B}$ will be represented as pairs

We have seem how to represent pairs in $\lambda\omega$.

$$\begin{aligned} \mathsf{Product} &: * \to * \to * \\ \mathsf{Product} &= \lambda \mathsf{A} : *.\ \lambda \mathsf{B} : *.\ (\forall \mathsf{C} : *.\ (\mathsf{A} \to \mathsf{B} \to \mathsf{C}) \to \mathsf{C}) \end{aligned}$$

So a conjunction $\mathsf{A} \land \mathsf{B}$ is represented as $\mathsf{Product\ A\ B}$

Rewritten in the general syntax of λC we have:

$$\mathsf{Product} = \lambda \mathsf{a} : *.\ \lambda \mathsf{b} : *.\ (\Pi \mathsf{c} : *.\ (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to \mathsf{c})$$

$$\mathsf{Product\ A\ B} = \Pi \mathsf{c} : *.\ (\mathsf{A} \to \mathsf{B} \to \mathsf{c}) \to \mathsf{c})$$

Now can we obtain λC rules for the rules of ND $\land_{\mathsf{I}}$ and $\land_{\mathsf{E1}}$ and $\land_{\mathsf{E2}}$

Rules for conjunction

We have to show that the following is derivable in λC:

$$\frac{\Gamma \vdash \mathcal{E}_1 : \mathcal{A} \qquad \Gamma, \vdash \mathcal{E}_2 : \mathcal{B}}{\Gamma \vdash ? : \mathsf{Product}\ \mathcal{A}\ \ \mathsf{B}} \qquad (\wedge_{\mathsf{I}})$$

$$\frac{\Gamma \vdash \mathcal{E} : \mathsf{Product}\ \mathcal{A}\ \mathcal{B}}{\Gamma \vdash ? : \mathcal{A}} \qquad (\wedge_{\mathsf{E1}})$$

$$\frac{\Gamma \vdash \mathcal{E} : \mathsf{Product}\ \mathcal{A}\ \mathcal{B}}{\Gamma \vdash ? : \mathcal{B}} \qquad (\wedge_{\mathsf{E2}})$$

Observe that rules for *implication* and for *universal quantification* were obtained more directly in λC.

Deriving $\wedge_I$ in λC

We have to show an inhabitant for the type $\Pi c : *. \, (\mathcal{A} \to \mathcal{B} \to c) \to c$ assuming that we have inhabitants for types $\mathcal{A}$ and $\mathcal{B}$.

$$\frac{\Gamma \vdash \mathcal{E}_1 : \mathcal{A} \qquad \Gamma, \vdash \mathcal{E}_2 : \mathcal{B}}{\Gamma \vdash ? : \Pi c : *. \, (\mathcal{A} \to \mathcal{B} \to c) \to c} \quad (\wedge_I)$$

We have already seem such an inhabitant and that requires only $\lambda\omega$. There that term was written as

$$\mathsf{pair} \; \langle \varphi \rangle \; \langle \psi \rangle \; e_1 \; e_2$$

Mapping to our case here we write:

$$\mathsf{pair} \; \langle \mathcal{A} \rangle \; \langle \mathcal{B} \rangle \; \mathcal{E}_1 \; \mathcal{E}_2$$

Which reduces to

$$\lambda c : *. \, \lambda f : \mathcal{A} \to \mathcal{B} \to c. \, f \; \mathcal{E}_1 \; \mathcal{E}_2$$

Exercise:

1. Obtain rules for $\wedge$ elimination. You can work on rules given for $\lambda\omega$
2. Obtain rules for $\vee$ introduction and rule for $\vee$ elimination. You can work on rules given for $\lambda\omega$

Encodings

These encodings can be done in $\lambda 2$

$$\begin{array}{lcl}
\bot & \equiv & \lambda x : *.x \\
(\neg\varphi)' & \equiv & \varphi' \to\bot \quad (\equiv \Pi x : \varphi'.\ \bot) \\
(\varphi \wedge \psi)' & \equiv & \Pi c : *(\varphi' \to \psi' \to c) \to c \\
(\varphi \vee \psi)' & \equiv & \Pi c : *.(\varphi' \to c) \to (\psi' \to c) \to c
\end{array}$$

Had we given encoding for the conectives we would nee $\lambda\omega$

$$\begin{array}{lcl}
\bot & \equiv & \lambda x : *.x \\
\neg & \equiv & \lambda z : *.\ z \to\bot \\
\wedge & \equiv & \lambda a : *.\ \lambda b : *.\ \Pi c : *.\ (a \to b \to c) \to c \\
\vee & \equiv & \lambda a : *.\ \lambda b : *.\ \Pi c : *.\ (a \to c) \to (b \to c) \to c
\end{array}$$

Encoding Existential Quantifier

Let's recall the existential elimination rule $\exists_E$:

$$\frac{\Gamma \vdash \exists x \in S.\, P(x) \qquad \Gamma \vdash \forall x \in S.\, (P(x) \Rightarrow A)}{\Gamma \vdash A} \quad (\exists_E)$$

The rules says:

- 1st premiss: if there is an x in S for which predicate P holds
- 2nd premiss: and if for any x in S: if P holds for x, then also A
- conclusion: then A holds (x does not occur free in A)

Encoding Existential

The premise $\forall x \in S.\ P(x) \Rightarrow A$ in λC is written as $\Pi x : S.\ (P\ x \to A)$

The fist premise with $\exists x \in S.\ P(x)$ is encoded as $\Pi a : *.\ ((\Pi x : S.\ (P\ x \to a) \to a)$

In words, what $\Pi a : *.\ ((\Pi x : S.\ (P\ x \to a) \to a)$ expresses:

For all a*:*
if we know that for all x *in* S*, it holds that* $P\ x$ *implies* a*, then* a *holds*

This codification of exists captures what the rule $\exists_E$ says about existential quantification

Now considering these codifications, the $\exists_E$ rule written with types of λC is:

$$\frac{\Gamma \vdash \Pi a : *.\ ((\Pi x : S.\ (P\ x \to a) \to a) \qquad \Gamma \vdash \Pi x : S.\ (P\ x \to A)}{\Gamma \vdash A}\ (\exists_E)$$

Exercise Construct a derivation of the rule above in λC. Assume that you start with the following Γ

$$S : *,\ P : S \to *,\ A : *,\ y : \Pi a : *.\ ((\Pi x : S.\ (P\ x \to a)) \to a)$$

Encoding Existential

Let's recall the existential elimination rule $\exists_\mathsf{I}$:

$$\frac{\Gamma \vdash b \in S \qquad \Gamma \vdash P(b)}{\Gamma \vdash \exists x \in S.\, P(x)} \qquad (\exists_\mathsf{I})$$

If we write this rule using λC we have

$$\frac{\Gamma \vdash b : S \qquad \Gamma \vdash P\ b}{\Gamma \vdash \Pi a : *.\ ((\Pi x : S.\ (P\ x \to a) \to a)} \qquad (\exists_\mathsf{I})$$

Exercise: Show that $\lambda a : *.\ \lambda v : (\Pi x : S.\ (P\ x \to a)).\ v\ b\ u$ is a proof object for the derivation of rule $\exists_\mathsf{I}$. You can start with the following Γ:

$$S : *,\ P : S \to *,\ b : S,\ u : P\ b$$

Proving first-order properties in λC

Let us prove some sequents of FOL using natural deduction:

Example: 1:

$$\forall x.P(x),\ \forall y.P(y) \Rightarrow Q(y)\ \vdash\ \forall z, Q(z)$$

Example: 2:

$$\exists x.P(x),\ \forall y.P(y) \Rightarrow Q(y) \wedge R(y)\ \vdash \exists z.R(z)$$

Example: 3:

$$\forall x.P(x)\ \vdash\ \neg(\exists y, \neg P(y))$$

$$\neg(\exists y, \neg P(y))\ \vdash\ \forall x.P(x)$$

Now let us consider how to encode the proofs in λC (using **Coq** as a tool to write the proof terms).

Natural numbers and induction in λC

Although we can represent inductive datatypes in λC, there are problems regarding the representation of the associated **induction principles**.

This is the motivation for extending λC with a primitive notion of indutive datatype in systems such as Coq and Agda.

Let us try to understand the problem: we will consider the definition of natural numbers in λC and its limitations.