Exponential objects

Let $\mathbf{C}$ be a category with products, and let A, B be objects. We say that the *exponential* of A and B exists if there is an object E and a natural isomorphism

$$\mathbf{C}(X \times A, B) \cong \mathbf{C}(X, E).$$

(natural in X). In this case, we usually write $E = B^A$, so that:

$$\mathbf{C}(X \times A, B) \cong \mathbf{C}(X, B^A).$$

Informally, B^A is a *space of functions* from A to B.

In other words, there is a bijective correspondence between morphisms $f : X \times A \to B$ and morphisms $f : X \to B^A$.

Exponential objects, continued

The definition of exponential objects can be understood in several equivalent ways.

More abstractly: A is *exponentiable* if the functor $F(X) = X \times A$ has a *right adjoint*. In this case, the right adjoint is written $G(B) = B^A$.

$$\mathbf{C}(X \times A, B) \cong \mathbf{C}(X, B^A).$$

$$\mathbf{C}(F(X), B) \cong \mathbf{C}(X, G(B)).$$

Exponential objects, continued

More concretely: An *exponential* for A and B is given by a pair (B^A, ϵ) where B^A is an object, $\epsilon : B^A \times A \to B$ is a morphism, and such that the following property holds:

For any object X and morphism $f : X \times A \to B$, there exists a *unique* morphism $h : X \to B^A$ such that

$$\begin{tikzcd} B^A \times A \arrow[r, "\epsilon"] & B \\ X \times A \arrow[u, dashed, "h\times A"] \arrow[ur, "f"'] & \end{tikzcd}$$

Exponential objects, if they exist, are unique up to isomorphism. We write $h = f^*$.

Cartesian-closed categories

Definition. A *cartesian-closed category* is a category with finite products (i.e., a products and a terminal object) and with exponential objects.

Part III: Lambda calculus

Recall: Types in programming

In computing, the *type* of a variable is the set of values that the variable can take. Examples of simple types are:

$$\mathbf{bit}, \mathbf{nat}, \mathbf{int}, \mathbf{string}, \ldots$$

We write $x : A$ to indicate that the variable x has type A.

Simple types can be combined by *type operations*. Examples:

$A \times B$:	Cartesian product (pairs of an A and a B)
$A + B$:	Disjoint union (either an A or a B)
$\mathbf{list}\, A$:	Type of lists of A's

We write (x, y) for a pair, and $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$ for the first and second component, respectively.

Higher-order functions

We write $f : A \to B$ for a *function* that takes inputs of type A and produces outputs of type B.

We can also regard $A \to B$ as a *type*, namely the type of all functions from A to B. This is called a *function space*.

A *higher order type* is a type where a function space occurs in a nested way, for example:

- a function that inputs another function: $(A \to B) \to C$,
- a function that outputs another function: $A \to (B \to C)$,
- a pair of two functions: $(A \to B) \times (C \to D)$.

A *higher order function* is a function of higher order type.

We need a language for manipulating higher order functions.

Example: Arithmetic expressions

Arithmetic expressions are made up from variables ($x, y, z \dots$), numbers ($1, 2, 3, \dots$), and operators ("$+$", "$-$", "$\times$" etc.)

The expression $x + y$ stands for the *result* of an addition (not an *instruction* to add, or the *statement* that something is being added).

We write

$$A = (x + y) \times z^2$$

One could write this as sequence of instructions:

let $w = x + y$, then let $u = z^2$, then let $A = w \times u$.

But such instructions would be cumbersome to manipulate, and algebraic laws impossible to state. Nested expressions are a powerful tool (which we take for granted).

Lambda calculus

The lambda calculus is an expression language for functions.
We normally write

Let f be the function defined by $f(x) = x^2$. Then consider $A = f(5)$,

In the lambda calculus we can just write

$$A = (\lambda x.x^2)(5).$$

The expression $\lambda x.x^2$ stands *for the function* that maps x to x^2 (as opposed to the *instruction* of squaring x, or the *statement* that x is being squared).

As for arithmetic, some of the power of the notation derives from the ability to nest expressions.

Examples

The composition operation $\circ$ of two functions:

We can write $f \circ g$ as $\lambda x.f(g(x))$.

We can write $C(f, g) = f \circ g$ as

$$\lambda f.\lambda g.f \circ g = \lambda f.\lambda g.\lambda x.f(g(x)).$$

Here, if $f : A \to B$ and $g : B \to C$, then $f \circ g : A \to C$, so the type of C is

$$C : (A \to B) \to ((B \to C) \to (A \to C))$$

Examples

The function mappair takes a function f and a pair (x, y), and returns $(f(x), f(y))$. It is an example of a higher-order function.

$$\begin{aligned} \text{mappair} &= \lambda f.\lambda p.(f(\pi_1 p), f(\pi_2 p)), \\ \text{mappair} &: (A \to B) \to ((A \times A) \to (B \times B)). \end{aligned}$$

Some do-it-yourself examples

Find lambda terms of the following types:

- $A \to A \times A$,
- $B \to (A \to A \times B)$,
- $(A \to C) \to (A \times B \to C)$,
- $A \to A \times B$,
- $(A \times (A \to B)) \to B$.

The Curry-Howard isomorphism

There is a fundamental connection between typed lambda calculus and intuitionistic propositional logic.

Translation:

- Basic types A, B, C are *propositional symbols*.
- Type operations $\times$, $+$, and $\to$ are *logical connectives* **and**, **or**, and $\Rightarrow$, respectively.

Proposition (Curry-Howard isomorphism): There exists a closed lambda term of a given type if and only if that type corresponds to a *tautology* of intuitionistic logic. Moreover, lambda terms correspond to *proofs*.

Examples

• $A \Rightarrow A$ **and** A	Provable: $\lambda x^A.(x, x)$
• $B \Rightarrow (A \Rightarrow A$ **and** $B)$	Provable: $\lambda x^B.\lambda y^A.(y, x)$
• $(A \Rightarrow C) \Rightarrow (A$ **and** $B \Rightarrow C)$	Provable: $\lambda f^{A \Rightarrow C}.\lambda p^{A\,\mathbf{and}\,B}.f(\pi_1(p))$
• $A \Rightarrow A$ **and** B	Not provable.
• $(A$ **and** $(A \Rightarrow B)) \Rightarrow B$	Provable: $\lambda x.(pi_2(x)(\pi_1(x)))$.

Cf. the *Brower-Heyting-Kolmogorov interpretation*: a proof of A **and** B is a pair of a proof of A and a proof of B. A proof of $A \Rightarrow B$ is a function that maps proofs of A to proofs of B.

The inference rules of intuitionistic logic

Assertions ("sequents") are of the form:

$$A_1, \ldots, A_n \vdash B,$$

meaning B is provable from assumptions $A_1, \ldots, A_n$.

$$\frac{}{\Gamma, A \vdash A} \qquad\qquad \frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \textbf{ and } B} \qquad \frac{\Gamma \vdash A \textbf{ and } B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \textbf{ and } B}{\Gamma \vdash B}$$

The typing rules of simply-typed lambda calculus

Assertions ("judgments") are of the form:

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : B,$$

meaning term M, with free variables $x_1, \ldots, x_n$ of respective types $A_1, \ldots, A_n$, is well-typed of type B.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash * : 1}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1(M) : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2(M) : B}$$

The evaluation of lambda terms

The basic computational rule of lambda calculus is β-reduction, which means, applying a function to an argument:

$$(\lambda x.M)N \to M[N/x],$$

$$\pi_1(M,N) \to M, \quad \pi_2(M,N) \to N.$$

We close these rules using *transitivity*, *reflexivity*, and *congruence*.

Theorem (Normalization): Every simply-typed lambda term reduces in a finite number of steps to a unique normal form.

Theorem (Subject reduction): If $\Gamma \vdash M : A$ is well-typed and $M \to^* N$, then $\Gamma \vdash N : A$.

With this, the lambda calculus is a (simple) programming language — see Lisp, ML, Haskell for real world examples.

The theory of $\beta\eta$ conversion

It makes sense to consider two lambda terms *equal* if they have the same normal form. Define β-equivalence, in symbols $=_\beta$ to the the smallest congruence relation containing β-reduction.

It also makes sense to consider so-called η-rules:

$$
\begin{array}{ll}
\lambda x.(Mx) =_\eta M, & \text{where } x \text{ is not free in } M, \\
(\pi_1 M, \pi_2 M) = M, & \text{where } M : A \times B, \\
* = M, & \text{where } M : 1.
\end{array}
$$

Let $=_{\beta\eta}$ be the smallest congruence relation containing β-reduction and η-equivalences.

The interpretation of simply-typed lambda calculus in **Set**

The simple type system can be interpreted in set theory, where a *type* is identified with a *set*.

- Basic types are interpreted as specific sets: $[\![\mathbf{bit}]\!] = \{0, 1\}$, $[\![\mathbf{nat}]\!] = \mathbb{N}$, etc.
- Type operations are interpreted as set operations:

$$\begin{array}{rcl} [\![1]\!] & = & 1, \\ [\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!], \\ [\![A \to B]\!] & = & [\![B]\!]^{[\![A]\!]}. \end{array}$$

- A context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is interpreted as a set:

$$[\![\Gamma]\!] \;=\; [\![A_1]\!] \times \ldots \times [\![A_n]\!].$$

- A typing judgement $\Gamma \vdash M : B$ is interpreted as a function:

$$[\![\Gamma \vdash M : B]\!] : [\![\Gamma]\!] \to [\![B]\!]$$

  defined *by recursion* on M.

The interpretation of simply-typed lambda calculus in cartesian-closed categories [Lambek]

Instead of *sets*, one can use the objects of any *cartesian-closed category*.

- Basic types are interpreted as specific *objects* $[\![A]\!]$, $[\![B]\!]$, etc.
- Type operations are interpreted *using the cartesian-closed structure*:

$$\begin{aligned} [\![1]\!] &= 1, \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!], \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]}. \end{aligned}$$

- A context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is interpreted as an *object*:

$$[\![\Gamma]\!] = [\![A_1]\!] \times \ldots \times [\![A_n]\!].$$

- A typing judgement $\Gamma \vdash M : B$ is interpreted as a *morphism*:

$$[\![\Gamma \vdash M : B]\!] : [\![\Gamma]\!] \to [\![B]\!]$$

  defined *by recursion* on M.

The interpretation of simply-typed lambda calculus in cartesian-closed categories, continued

$$
\begin{array}{lcl}
[\![\Gamma, x : A \vdash x : A]\!] & = & [\![\Gamma]\!] \times [\![A]\!] \xrightarrow{\pi_2} [\![A]\!] \\
[\![\Gamma \vdash * : 1]\!] & = & [\![\Gamma]\!] \xrightarrow{*} 1 \\
[\![\Gamma \vdash \lambda x.M : A \to B]\!] & = & [\![\Gamma]\!] \xrightarrow{[\![\Gamma, x:A \vdash M:B]\!]^*} [\![B]\!]^{[\![A]\!]} \\
[\![\Gamma \vdash MN : B]\!] & = & [\![\Gamma]\!] \xrightarrow{\langle [\![\Gamma \vdash M:A \to B]\!], [\![\Gamma \vdash N:A]\!] \rangle} [\![B]\!]^{[\![A]\!]} \times [\![A]\!] \xrightarrow{\epsilon} [\![B]\!]. \\
[\![\Gamma \vdash (M, N) : A \times B]\!] & = & [\![\Gamma]\!] \xrightarrow{\langle [\![\Gamma \vdash M:A]\!], [\![\Gamma \vdash N:B]\!] \rangle} [\![A]\!] \times [\![B]\!] \\
[\![\Gamma \vdash \pi_1(M) : A]\!] & = & [\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash M:A \times B]\!]} [\![A]\!] \times [\![B]\!] \xrightarrow{\pi_1} [\![A]\!] \\
[\![\Gamma \vdash \pi_2(M) : B]\!] & = & [\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash M:A \times B]\!]} [\![A]\!] \times [\![B]\!] \xrightarrow{\pi_2} [\![B]\!]
\end{array}
$$

Theorem 1. The interpretation of the simply-typed lambda calculus in cartesian-closed categories is *sound*. In other words, if $\Gamma \vdash M = N : A$, then $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!]$. (Easy, by induction).

Theorem 2. The interpretation of the simply-typed lambda calculus in cartesian-closed categories is *complete*. In other words, if $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!]$ for *all* interpretations in *all* cartesian-closed categories, then $\Gamma \vdash M = N : A$.

Theorem 3. The simply-typed lambda calculus is an *internal language* for cartesian-closed categories.

(Roughly: there is a one-to-one correspondence between models of the lambda calculus and cartesian-closed categories).

The term model

Fix a set of basic types. The *term model* of the simply-typed lambda calculus is a category Λ, constructed as follows:

- The *objects* are types.
- A *morphism* $f : A \to B$ is a $\beta\eta$-equivalence class of typing judgements of the form $x : A \vdash M : B$.

Theorem 4. The term model Λ is a cartesian-closed category. Moreover, for any cartesian-closed category $\mathbf{C}$, there is a bijective correspondence between:

- Maps assigning objects of $\mathbf{C}$ to basic types;
- Interpretations of the lambda calculus in $\mathbf{C}$; and
- Cartesian closed functors $F : \Lambda \to \mathbf{C}$.

The Curry-Howard-Lambek isomorphism

By the results of the previous slides Λ is the *free* cartesian-closed category, and cartesian-closed categories and the lambda calculus (and therefore intuitionistic propositional logic) are essentially the same.

$$\begin{tikzcd}
& \text{Lambda calculus} \arrow[dl, Leftrightarrow] \arrow[dr, Leftrightarrow] & \\
\text{Intuitionistic propositional logic} \arrow[rr, Leftrightarrow] & & \text{Cartesian-closed categories}
\end{tikzcd}$$

Extensions of the Curry-Howard isomorphism

The Curry-Howard isomorphism gives a basic connection between *programming languages* and *logic*. This connection can be usefully extended in both directions:

- given a programming language feature, one can ask for its logical meaning.
- Given a logical feature, one can ask for its computational meaning.

Examples:

Logic	**Programming**
A **or** B	Sum type $A + B$
$\forall$ quantifier	Polymorphism: $\lambda x.x : \forall A.A \to A$
$\exists$ quantifier	Data abstraction: $\exists D.(A \times D \to B) \times D$
Classical logic A **or** $\neg A$	Continuations
Type theory	Dependently typed programming
Topos logic	Set comprehension
...	...