Symbolic Expressions and Variable Binding
Lecture 2

Masahiko Sato

Graduate School of Informatics, Kyoto University

September 6–10, 2010

Plan of the 5 lectures

1. Overview
2. Traditional definition of Lambda terms
3. Lambda terms by de Bruijn indices
4. Lambda terms as abstract data type
5. Derivations as abstract data type

Plan of this lecture

- Objects of the first kind
- Objects of the second kind
- Natural numbers
- Binary trees
- Abstract syntax
- Equivariance
- Traditional lambda expressions

Objects of the first kind

Objects of the first kind are created by the fundamental principle of object creation:

> *Every object $\boldsymbol{a}$ is created from already created $\boldsymbol{n}$ objects $\boldsymbol{a_1, \ldots, a_n}$ $(\boldsymbol{n \geq 0})$ by applying a creation method M.*

We can visualize this *act* of creation by the following figure:

$$\frac{\boldsymbol{a_1 \quad \cdots \quad a_n}}{\boldsymbol{a}}\ M$$

or, by the equation:

$$\boldsymbol{a} = M\,(\boldsymbol{a_1, \ldots, a_n})$$

Objects of the first kind (cont.)

Mathematical objects of the first kind are constructed by the fundamental priciple of object creation:

- An object of the first kind is created from finitely many already created objects of the first kind.
- The creation is done by applying a creation method to existing objects.
- Both the creation method and the created object belongs to a specific class.
- The class is called the mother class of the created object.
- Thus, any object is created as an instance of its mother class.
- The equality relation ($=$) on objects of the first kind is called the equality of the first kind.

Objects of the first kind (cont.)

Objects of the first kind have the following nice properties.

- For any object a and any mother class C, it is decidable whether $a : C$ or not.
- Primitive recursive computation on objects of any mother class is possible.
- Induction principle on objects of any mother class can be given uniformly.
- A method is an object of the first kind. (In contrast, a function is usually extensional, and is defined as an object of the second kind.)
- A class is an object of the first kind. It is an instance of the mother class $\langle \texttt{Class} \rangle$.
- In particular, we have $\langle \texttt{Class} \rangle : \langle \texttt{Class} \rangle$.

Objects of the first kind (cont.)

Objects of the first kind have the following nice properties.

- For any object a and any mother class C, it is decidable whether $a : C$ or not.
- Primitive recursive computation on objects of any mother class is possible.
- Induction principle on objects of any mother class can be given uniformly.
- A method is an object of the first kind. (In contrast, a function is usually extensional, and is defined as an object of the second kind.)
- A class is an object of the first kind. It is an instance of the mother class $\langle \texttt{Class} \rangle$.
- In particular, we have $\langle \texttt{Class} \rangle : \langle \texttt{Class} \rangle$.

We are developing a programming language, Z, which can be used to create, compute and reason about objects of the first kind.

Objects of the second kind

Let C be a class whose members are objects of the first kind, and let $=_2$ be a (partial) equivalence relation on C.

We can obtain objects of the second kind by identifying a and b in C if $a =_2 b$. When $=_2$ is a partial equivalence relation, an object a of the first kind in C is considered to be an object of the second kind if $a =_2 a$ holds.

In this setting, functions and relations on these objects must be defined so that the equality $=_2$ becomes congruence relation with respect to these functions and relations.

Well-definedness of these functions and relations are sometimes nontrivial.

Also, inductive arguments are not as smooth as for objects of the first kind, or even impossible.

Natural numbers

We define natural numbers as instances of a mother class $\langle \texttt{Nat} \rangle$.

$$\frac{}{(\texttt{Nat/zro}) : \langle \texttt{Nat} \rangle}\ \texttt{Nat/zro} \qquad \frac{\boldsymbol{n} : \langle \texttt{Nat} \rangle}{(\texttt{Nat/suc}\ \boldsymbol{n}) : \langle \texttt{Nat} \rangle}\ \texttt{Nat/suc}$$

- The first creation method `Nat/zro`, creates an instance `(Nat/zro)` from 0 already created objects.
- We can read off the above fact, simply by looking at the created object `(Nat/zro)`.
- The second creation method `Nat/suc`, creates an instance $(\texttt{Nat/suc}\ \boldsymbol{n})$ from 1 already created object $\boldsymbol{n}$, provided that $\boldsymbol{n}$ satisfies the side condition: $\boldsymbol{n}$ is an instance of $\langle \texttt{Nat} \rangle$.
- The premise of the method $\boldsymbol{n} : \langle \texttt{Nat} \rangle$ express the above side condition.

Natural numbers (cont.)

We will write these methods in the following concise form.

$$\frac{}{(\texttt{zro}) : \langle\texttt{Nat}\rangle}\ \texttt{zro} \qquad\qquad \frac{\boldsymbol{n} : \langle\texttt{Nat}\rangle}{(\texttt{suc}\ \boldsymbol{n}) : \langle\texttt{Nat}\rangle}\ \texttt{suc}$$

It is possible to display natural numbers in tree forms. For example, $\mathbf{3} =$ `(suc (suc (suc (zro))))` can be displayed as follows.

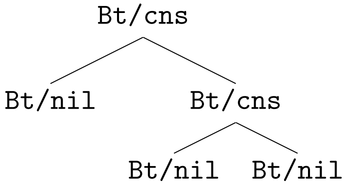
$$\begin{array}{c} \text{Nat/suc} \\ | \\ \text{Nat/suc} \\ | \\ \text{Nat/suc} \\ | \\ \text{Nat/zro} \end{array}$$

Binary trees

We define binary trees as instances of a mother class $\langle \texttt{Bt} \rangle$.

$$\frac{}{(\texttt{nil}) : \langle \texttt{Bt} \rangle}\ \texttt{nil} \qquad \frac{\boldsymbol{s} : \langle \texttt{Bt} \rangle \quad \boldsymbol{t} : \langle \texttt{Bt} \rangle}{(\texttt{cns}\ \boldsymbol{s}\ \boldsymbol{t}) : \langle \texttt{Bt} \rangle}\ \texttt{cns}$$

It is, of course, possible to display binary trees in tree forms. For example, `(cns (nil) (cns (nil) (nil)))` can be displayed as follows.



Lists

We define lists (of natural numbers) as instances of a mother class $\langle\texttt{List}\rangle$.

$$\frac{}{(\texttt{nil}) : \langle\texttt{List}\rangle}\ \texttt{nil} \qquad \frac{\boldsymbol{a} \quad \boldsymbol{L} : \langle\texttt{List}\rangle}{(\texttt{cns}\ \boldsymbol{a}\ \boldsymbol{L}) : \langle\texttt{List}\rangle}\ \texttt{cns}$$

The second method `cns` (cons) is a binary method where its first argmuent can be any already created object $\boldsymbol{a}$, but the second argument $\boldsymbol{L}$ must be a $\langle\texttt{List}\rangle$.

Lists (cont.)

$$\frac{}{(\texttt{nil}) : \langle\texttt{List}\rangle}\ \texttt{nil} \qquad\qquad \frac{\boldsymbol{a} \quad \boldsymbol{L} : \langle\texttt{List}\rangle}{(\texttt{cns}\ \boldsymbol{a}\ \boldsymbol{L}) : \langle\texttt{List}\rangle}\ \texttt{cns}$$

It is, of course, possible to display binary trees in tree forms. For example, `(cns 1 (cns 2 (nil)))` can be displayed as follows.

$$\Tree [.\texttt{List/cns} \texttt{1} [.\texttt{List/cns} \texttt{2} \texttt{List/nil}]]$$

Abstract syntax

- McCarthy (1963) introduced the notion of abstact syntax.
- Abstract syntax deals with syntactic objects as objects of abstract data types.
- It is possible to compute and reason about objects only by means of functions exported from the data types.
- These functions are usually classified into: constructors, recognizers and selectors.
- Hence, objects of the first kind belonging to a same mother class can be presented as an abstract data type.
- Objects of the first kind are abstract in this sense.

Abstract syntax vs. axiom system

An abstract data type given by abstract syntax is similar to an axiomatic system.

An axiomatic system specifies a mathematical structure abstractly.

Take, for example, Peano Arithmetic. It specifies the structure of natural numbers, in terms of the nullary function `zro` and the unary function `suc`, and abstractly specifies the structure in terms of Peano's axioms.

There are many (although isomorphic) concrete implementations (models) of the axiom system.

Similarly, the class $\langle$`Nat`$\rangle$ is an absract data type whose structure is given to the users of the data type only through the names of primitive functions like `zro` and `suc`, and their arities.

Hence, the implementor of the data type can hide the details of implementation from the users.

Computation on Abstract syntax

In Z, we have the case expressions which can be used to define recursive functions on objects.

When the value of e is a natural number:

```
(case e
  ((zro) ... )
  ((suc n) ... n ... ))
```

When the value of e is a binary tree:

```
(case e
  ((nil) ... )
  ((cns s t) ... s t ... ))
```

Computation on Abstract syntax

In Z, we have the case expressions which can be used to define recursive functions on objects.

When the value of e is a natural number:

```
(case e
  ((zro) ... )
  ((suc n) ... n ... ))
```

When the value of e is a binary tree:

```
(case e
  ((nil) ... )
  ((cns s t) ... s t ... ))
```

[demo]

Traditional lambda expressions

We define traditional lambda expressions as instances of a mother class $\langle\texttt{Txp}\rangle$.

$$\frac{x : \langle\texttt{Nat}\rangle}{(\texttt{var}\ x) : \langle\texttt{Txp}\rangle}\ \texttt{var} \qquad \frac{M : \langle\texttt{Txp}\rangle \quad N : \langle\texttt{Txp}\rangle}{(\texttt{app}\ M\ N) : \langle\texttt{Txp}\rangle}\ \texttt{app}$$

$$\frac{x : \langle\texttt{Nat}\rangle \quad M : \langle\texttt{Txp}\rangle}{(\texttt{lam}\ x\ M) : \langle\texttt{Txp}\rangle}\ \texttt{lam}$$

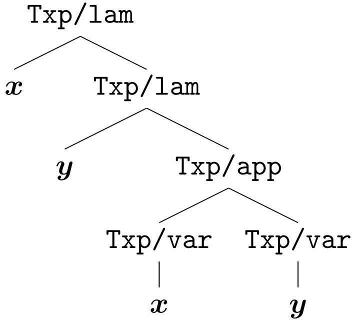
An example, where x and y are distinct natural numbers.

$(\texttt{lam}\ x\ (\texttt{lam}\ y\ (\texttt{app}\ (\texttt{var}\ x)\ (\texttt{var}\ y))))$

Traditional lambda expressions (cont.)

(lam x (lam y (app (var x) (var y))))

The tree form of this expression is:



Group action

A group G acts on a set X if there is a group action map:

$$\cdot : G \times X \to X$$

with the following properties.

1. $1_G \cdot x = x$ for all $x \in X$.
2. $ab \cdot x = a \cdot (b \cdot x)$ for all $a, b \in G$ and $x \in X$.

Note that each $a \in G$ induces a bijection:

$$a^* : X \to X$$

such that $a^*(x) = a \cdot x$ $(x \in X)$.
Also, $G^* := \{a^* \mid a \in G\}$ becomes a group isomorphic to G under the group operation $(\circ)$ defined by
$(a^* \circ b^*)(x) := a^*(b^*(x)) = a \cdot (b \cdot x) = ab \cdot x$.

Finite permutations

We will mainly consider the group G of finite permutations on $\langle \texttt{Nat} \rangle$. A bijection $\rho : V \to V$ is a finite permutation on $\langle \texttt{Nat} \rangle$ if it fixes all but finitely many $x : \langle \texttt{Nat} \rangle$. The group operation is defined by $(\rho \circ \sigma)(x) = \rho(\sigma(x))$. G acts on $\langle \texttt{Nat} \rangle$ in a natural way.

Each element ρ of G can be expressed as:

$$\rho = [x_{\pi(1)}, \ldots, x_{\pi(n)}/x_1, \ldots, x_n]$$

where π is a permutaion on the set $\{1, \ldots, n\}$ and $x_1, \ldots, x_n$ are n distinct natural numbers. ρ is a bijection $\rho : V \to V$ such that $\rho(x)$ is $x_{\pi(i)}$ if $x = x_i$ and x otherwise.

If x and y are distinct, then the permutaion $[y, x/x, y]$ is called a swap and we write $(y /\!\!/ x)$ for it. A swap is its own inverse, and any finite permutation can be written as a product of swaps.

Equivariance

Let G be a group acting on two sets X and Y.
Then a function $f : X \to Y$ is a equivariant map if

$$f(a \cdot x) = a \cdot f(x)$$

holds for all $a \in G$ and $x \in X$.

Equivariance

Let G be a group acting on two sets X and Y.
Then a function $f : X \to Y$ is a equivariant map if

$$f(a \cdot x) = a \cdot f(x)$$

holds for all $a \in G$ and $x \in X$.

We will take as G the group $\mathbf{Perm}$ of finite permutations on variables, and analyze the structure of the traditional lambda expressions.

Action of Perm on $\langle\texttt{List}\rangle$ and $\langle\texttt{Txp}\rangle$

We can define two swap functions, one on $\langle\texttt{List}\rangle$ and the other on $\langle\texttt{Txp}\rangle$.

$$\texttt{List/swap} : \langle\texttt{Nat}\rangle\ \langle\texttt{Nat}\rangle\ \langle\texttt{List}\rangle \to \langle\texttt{List}\rangle$$

```
(defun List/swap (x y L)
  (case L
    ((nil) [])
    ((cns z L)
     (List/cns
      (if (=? z x) y
        (if (=? z y) x
          z))
      (List/swap x y L)))))
```

Action of Perm on ⟨List⟩ and ⟨Txp⟩

We can define two swap functions, one on ⟨List⟩ and the other on ⟨Txp⟩.

$$\texttt{Txp/swap} : \langle\texttt{Nat}\rangle\ \langle\texttt{Nat}\rangle\ \langle\texttt{Txp}\rangle \rightarrow \langle\texttt{Txp}\rangle$$

```
(defun Txp/swap (x y M)
  (case M
    ((var z)
     (if (=? z x) (Txp/var y)
       (if (=? z y) (Txp/var x)
         (Txp/var z))))
    ((app M1 M2)
     (Txp/app (Txp/swap x y M1) (Txp/swap x y M2)))
    ((lam z M1)
     (Txp/lam
      (if (=? z x) y (if (=? z y) x z))
      (Txp/swap x y M1)))))
```

Free variables

We define the function:

$$\mathtt{FV} : \langle \mathtt{Txp} \rangle \longrightarrow \langle \mathtt{List} \rangle$$

as follows.

```
(defun FV (M)
  (case M
    ((var M) [M])
    ((app M N) (append (FV M) (FV N)))
            ((lam x M) (remove x (FV M)))))
```

Free variables (cont.)

`append` is defined as follows.

```
(defun append (L1 L2)
  (case L1
    ((nil) L2)
    ((cns x L1) (List/cns x (append L1 L2)))))
```

`remove` is defined as follows.

```
(defun remove (x L)
  (case L
    ((nil) L)
    ((cns y L)
     (let ((L1 (remove x L)))
       (if (=? x y) L1 (List/cns y L1))))))
```

Free variables (cont.)

We also define `in?` for later use.

$$\texttt{in?} : \langle\texttt{object}\rangle\ \langle\texttt{List}\rangle \to \langle\texttt{bool}\rangle$$

where $\langle\texttt{object}\rangle$ is the class of all objects and $\langle\texttt{bool}\rangle$ is the class consisting of `true` (= `nil` = `[]`) and `false` (= `()`).

```
(defun in? (x L)
  "Check if <object> x is in <List> L."
  (case L
    ((nil) nil)
    ((cns y L) (or (=? x y) (in? x L)))))
```

Alpha equivalence

We define an equivariant function:

$$\texttt{Txp/=?} : \langle\texttt{Txp}\rangle\ \langle\texttt{Txp}\rangle \rightarrow \langle\texttt{bool}\rangle$$

as follows.

```
(defun Txp/=? (M N)
  (case M
    ((var x)
     (case N
       ((var y) (=? x y))))
    ((app M1 M2)
     (case N
       ((app N1 N2) (and (Txp/=? M1 N1) (Txp/=? M2 N2)))))
    ((lam x M1)
     (case N
       ((lam y N1) (Txp/=? (Txp/swap x y M1) N1))))))
```

Alpha equivalence (cont.)

Two traditional lambda expressions M and N are alpha equivalent (written, $M =_\alpha N$) if `(Txp/=? M N) = ()` (true). $=_\alpha$ enjoys the following properties.

1. If $y \notin$ `(FV M)`, then `(lam x M)` $=_\alpha$ `(lam y (y//x)M)`.
2. (refl) $M =_\alpha M$.
3. (symm) If $M =_\alpha N$, then $N =_\alpha M$.
4. (trans) If $M =_\alpha N$ and $N =_\alpha P$, then $M =_\alpha P$.
5. If $M =_\alpha N$, then `(lam x M)` $=_\alpha$ `(lam x N)`.
6. If $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$, then `(app M1 N1)` $=_\alpha$ `(app M2 N2)`.

Remark 1: It is possible to inductively define $=_\alpha$ as the least relation having the above properties. Our definition is more basic than such a definition, since it gives a primitive recursive function to decide the alpha equivalence.

Alpha equivalence (cont.)

Two traditional lambda expressions M and N are alpha equivalent (written, $M =_\alpha N$) if (`Txp/=?` M N) $=$ `()` (true). $=_\alpha$ enjoys the following properties.

1. If $y \notin$ (`FV` M), then (`lam` x M) $=_\alpha$ (`lam` y $(y /\!\!/ x)M$).
2. (refl) $M =_\alpha M$.
3. (symm) If $M =_\alpha N$, then $N =_\alpha M$.
4. (trans) If $M =_\alpha N$ and $N =_\alpha P$, then $M =_\alpha P$.
5. If $M =_\alpha N$, then (`lam` x M) $=_\alpha$ (`lam` x N).
6. If $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$, then (`app` M_1 N_1) $=_\alpha$ (`app` M_2 N_2).

Remark 2: In the standard definition, item 1 above is expressed as:

1. If $y \notin$ (`FV` M), then (`lam` x M) $=_\alpha$ (`lam` y $[y/x](M)$).

But, this requires to define substitution ($[\ /\ ]()$) before defining alpha equivalence.

Alpha equivalence (cont.)

Two traditional lambda expressions M and N are alpha equivalent (written, $M =_\alpha N$) if `(Txp/=? M N) = ()` (true). $=_\alpha$ enjoys the following properties.

1. If $y \notin$ (FV M), then (lam x M) $=_\alpha$ (lam y $(y /\!\!/ x)M$).
2. (refl) $M =_\alpha M$.
3. (symm) If $M =_\alpha N$, then $N =_\alpha M$.
4. (trans) If $M =_\alpha N$ and $N =_\alpha P$, then $M =_\alpha P$.
5. If $M =_\alpha N$, then (lam x M) $=_\alpha$ (lam x N).
6. If $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$, then (app M_1 N_1) $=_\alpha$ (app M_2 N_2).

Remark 3: Swap function $(\cdot /\!\!/ \cdot)\cdot$ is an equivariant function, but it is (probably) not possible to define substitution function $[\cdot / \cdot](\cdot)$ as an equivariant function.

Substitution

We now know that $=_\alpha$ is an equivalence relation on $\langle\texttt{Txp}\rangle$ compatible with the three creation methods: `Txp/var`, `Txp/app` and `Txp/lam`.

So, from now on, we will regard instances of $\langle\texttt{Txp}\rangle$ as objects of the second kind and define a function `Txp/subst` which performs substitution operation on $\langle\texttt{Txp}\rangle$.

$$\texttt{Txp/subst} : \langle\texttt{Txp}\rangle\langle\texttt{Nat}\rangle\langle\texttt{Txp}\rangle \to \langle\texttt{Txp}\rangle$$

We will write $[N/x](M)$ for (the value of) $(\texttt{Txp/subst}\ N\ x\ M)$.

This function must enjoy the property:

If $N_1 =_\alpha N_2$ and $M_1 =_\alpha M_2$, then $[N_1/x](M_1) =_\alpha [N_2/x](M_2)$.

Substitution (cont.)

```
(defun subst (N x M)
  (case M
    ((var y) (if (=? x y) N M))
    ((app M1 M2) (Txp/app (subst N x M1) (subst N x M2)))
    ((lam y M1)
     (if (=? x y) M
       (let ((m1 (FV M1)) (n (FV N)))
         (if (and (in? x m1) (in? y n))
             (let* ((z ((fresh (append m1 n)))
                    (M2 (Txp/swap y z M1)))
               (Txp/lam z (subst N x M2)))
           (Txp/lam y (subst N x M1))))))))
```

Substitution (cont.)

```
(defun fresh (L)
  (case L
    ((nil) 0)
    ((cns x L)
     (max (1+ x) (fresh L)))))
```

This is not an equivariant function.