Symbolic Expressions and Variable Binding
Lecture 3

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 the talk

- *de Bruijn indices* are used to represent lambda expressions by means of nameless binders.
- We will define lambda expressions using de Bruijn indices in two steps.

Plan of the talk

- *de Bruijn indices* are used to represent lambda expressions by means of nameless binders.
- We will define lambda expressions using de Bruijn indices in two steps.
- In the first step we define the mother class ⟨`Dxp`⟩ of de Bruijn expressions which may contain *free* indices. This corresponds to the *raw terms* of de Bruijn's expressions in the usual presentation.

Plan of the talk

- *de Bruijn indices* are used to represent lambda expressions by means of nameless binders.
- We will define lambda expressions using de Bruijn indices in two steps.
- In the first step we define the mother class ⟨`Dxp`⟩ of de Bruijn expressions which may contain *free* indices. This corresponds to the *raw terms* of de Bruijn's expressions in the usual presentation.
- In the second step, we define the mother class ⟨`dxp`⟩ consisting of *closed* de Buijn expressions containing no free indices. This class *implements* lambda expressions as objects of the first kind.

Plan of the talk

- *de Bruijn indices* are used to represent lambda expressions by means of nameless binders.
- We will define lambda expressions using de Bruijn indices in two steps.
- In the first step we define the mother class ⟨`Dxp`⟩ of de Bruijn expressions which may contain *free* indices. This corresponds to the *raw terms* of de Bruijn's expressions in the usual presentation.
- In the second step, we define the mother class ⟨`dxp`⟩ consisting of *closed* de Buijn expressions containing no free indices. This class *implements* lambda expressions as objects of the first kind.
- Finally, we compare ⟨`dxp`⟩ with the class ⟨`Txp`⟩ of traditional expressions.

The class $\langle\texttt{Dxp}\rangle$

The mother class $\langle\texttt{Dxp}\rangle$ (de Bruijn lambda expression) has the following creation methods:

$$\frac{x : \langle\texttt{Nat}\rangle}{(\texttt{var}\ \ x) : \langle\texttt{Dxp}\rangle}\ \texttt{var} \qquad \frac{i : \langle\texttt{Nat}\rangle}{(\texttt{idx}\ \ i) : \langle\texttt{Dxp}\rangle}\ \texttt{idx}$$

$$\frac{M : \langle\texttt{Dxp}\rangle \quad N : \langle\texttt{Dxp}\rangle}{(\texttt{app}\ M\ \ N) : \langle\texttt{Dxp}\rangle}\ \texttt{app} \qquad \frac{M : \langle\texttt{Dxp}\rangle}{(\texttt{lam}\ M) : \langle\texttt{Dxp}\rangle}\ \texttt{lam}$$

Ramark 1 The creation method idx creates an index which replaces a bound variable in traditional lambda expression.

The class $\langle \texttt{Dxp} \rangle$

The mother class $\langle \texttt{Dxp} \rangle$ (de Bruijn lambda expression) has the following creation methods:

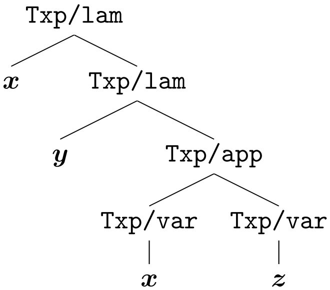
$$\frac{x : \langle \texttt{Nat} \rangle}{(\texttt{var}\ \ x) : \langle \texttt{Dxp} \rangle}\ \texttt{var} \qquad \frac{i : \langle \texttt{Nat} \rangle}{(\texttt{idx}\ \ i) : \langle \texttt{Dxp} \rangle}\ \texttt{idx}$$

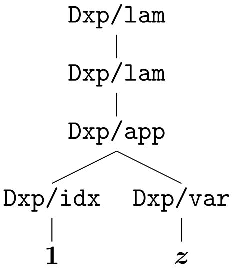
$$\frac{M : \langle \texttt{Dxp} \rangle \quad N : \langle \texttt{Dxp} \rangle}{(\texttt{app}\ M\ \ N) : \langle \texttt{Dxp} \rangle}\ \texttt{app} \qquad \frac{M : \langle \texttt{Dxp} \rangle}{(\texttt{lam}\ M) : \langle \texttt{Dxp} \rangle}\ \texttt{lam}$$

Ramark 2 Unlike traditional case, `lam` method does not have the argument for binding variables.

The class ⟨Dxp⟩ (cont.)

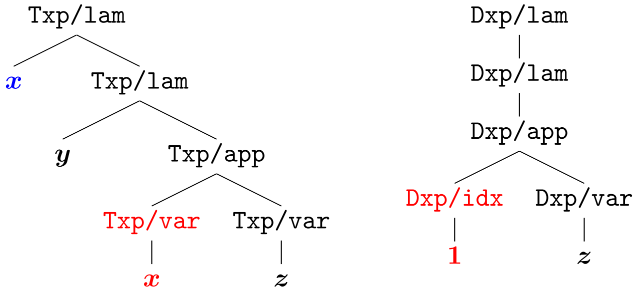
$$\boldsymbol{\lambda x[\lambda y[x \cdot z]]}$$





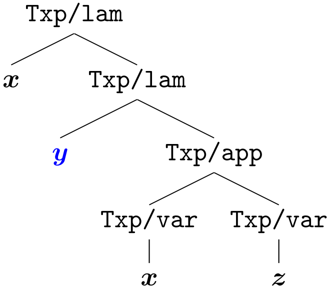
The class ⟨Dxp⟩ (cont.)

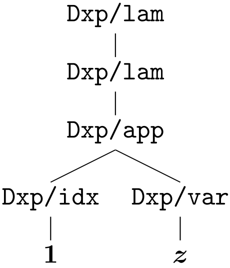
$$\boldsymbol{\lambda x[\lambda y[x \cdot z]]}$$



The class $\langle$Dxp$\rangle$ (cont.)

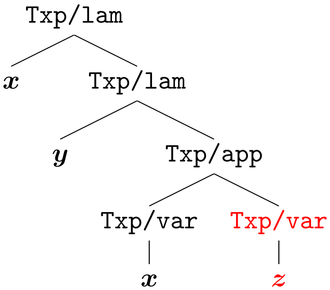
$$\boldsymbol{\lambda x[\lambda y[x \cdot z]]}$$

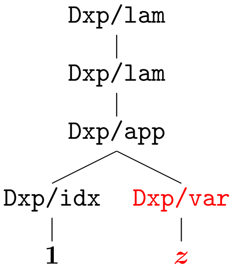




The class ⟨Dxp⟩ (cont.)

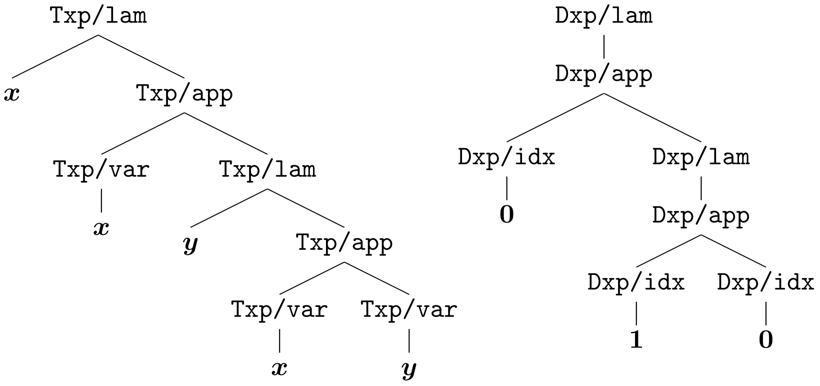
$$\boldsymbol{\lambda x[\lambda y[x \cdot z]]}$$





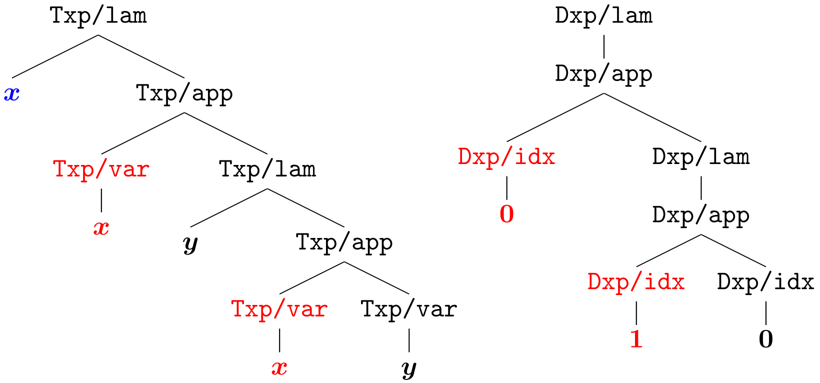
The class ⟨Dxp⟩ (cont.)

$$\boldsymbol{\lambda x[x \cdot \lambda y[x \cdot y]]}$$



The class ⟨Dxp⟩ (cont.)

$$\boldsymbol{\lambda x[x \cdot \lambda y[x \cdot y]]}$$



Action of Perm on ⟨Dxp⟩

We can define the swap function:

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

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

Remark 1 All the functions we introduce in Lecture 3 are *equivariant* functions.

Action of Perm on ⟨Dxp⟩

We can define the swap function:

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

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

Remark 2 We will not use the Dxp/swap function anymore.

The function Dxp/Closed?

We define the function:

$$\texttt{Dxp/Closed?} : \langle\texttt{Dxp}\rangle\ \langle\texttt{Nat}\rangle \to \langle\texttt{bool}\rangle$$

which checks if a given $\langle\texttt{Dxp}\rangle$ is *closed* at a given level or not.

```
(defun Dxp/Closed? (M i)
  (case M
    ((var x) true)
    ((idx j) (<? j i))
    ((app M N)
     (and (Dxp/Closed? M i) (Dxp/Closed? N i)))
    ((lam M) (Dxp/Closed? M (1+ i)))))
```

The function `Dxp/closed?`

We define the function:

$$\texttt{Dxp/closed?} : \langle\texttt{Dxp}\rangle \to \langle\texttt{bool}\rangle$$

which checks if a given $\langle\texttt{Dxp}\rangle$ is *closed* or not. A $\langle\texttt{Dxp}\rangle$ is closed iff every index in it is bound by a `Dxp/lam`.

```
(defun Dxp/closed? (M)
  (Dxp/Closed? M 0))
```

The function `Dxp/closed?` (cont.)

Of the five expressions below, only the last one is not closed.
$\boldsymbol{\lambda} \boldsymbol{wxyz}[z], \boldsymbol{\lambda} \boldsymbol{wxyz}[y], \boldsymbol{\lambda} \boldsymbol{wxyz}[x], \boldsymbol{\lambda} \boldsymbol{wxyz}[w], \boldsymbol{\lambda} \boldsymbol{wxyz}[v]$

The function `Dxp/closed?` (cont.)

Of the five expressions below, only the last one is not closed.
$\boldsymbol{\lambda wxyz[z], \lambda wxyz[y], \lambda wxyz[x], \lambda wxyz[w], \lambda wxyz[v]}$

```
Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam
   |          |          |          |          |
Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam
   |          |          |          |          |
Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam
   |          |          |          |          |
Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam    Dxp/lam
   |          |          |          |          |
Dxp/idx    Dxp/idx    Dxp/idx    Dxp/idx    Dxp/idx
   |          |          |          |          |
   0          1          2          3          4
```

The function Dxp/close

We define two functions

$$\texttt{Dxp/close} : \langle\texttt{Nat}\rangle\ \langle\texttt{Dxp}\rangle \rightarrow \langle\texttt{Dxp}\rangle$$
$$\texttt{Dxp/Close} : \langle\texttt{Nat}\rangle\ \langle\texttt{Nat}\rangle\ \langle\texttt{Dxp}\rangle \rightarrow \langle\texttt{Dxp}\rangle$$

```
(defun Dxp/close (x M)
  (Dxp/Close x 0 M))

(defun Dxp/Close (x i M)
  (case M
    ((var y) (if (=? x y) (Dxp/idx i) M))
    ((idx j) M)
    ((app M N)
     (Dxp/app (Dxp/Close x i M) (Dxp/Close x i M)))
    ((lam M) (Dxp/lam (Dxp/Close x (1+ i) M)))))
```

The class $\langle$Dxp0$\rangle$

We can define the class $\langle$Dxp0$\rangle$ as a *subclass* of $\langle$Dxp$\rangle$ consisting of closed instances of $\langle$Dxp$\rangle$.

$$\texttt{Dxp0?} : \langle\texttt{object}\rangle \to \langle\texttt{bool}\rangle$$

```
(defun Dxp0? (M)
  "Check if a given <object> M is an instance of <Dxp>."
  (and (Dxp? M)
       (Dxp/closed? M)))
```

However, this class is not a mother class.

So, we construct a mother class which is an isomorphic but disjoint copy of $\langle$Dxp0$\rangle$ in the next slide.

The class ⟨Dxp0⟩

We can define the class ⟨Dxp0⟩ as a *subclass* of ⟨Dxp⟩ consisting of closed instances of ⟨Dxp⟩.

$$\texttt{Dxp0?} : \langle\texttt{object}\rangle \to \langle\texttt{bool}\rangle$$

```
(defun Dxp0? (M)
  "Check if a given <object> M is an instance of <Dxp>."
  (and (Dxp? M)
       (Dxp/closed? M)))
```

However, this class is not a mother class.

So, we construct a mother class which is an isomorphic but disjoint copy of ⟨Dxp0⟩ in the next slide.

[demo]

The class $\langle \texttt{dxp} \rangle$

The class $\langle \texttt{Dxp} \rangle$ contains *open* (that is, non-closed) expressions which do not represent valid lambda expressions. So, we introduce the class $\langle \texttt{dxp} \rangle$ which is obtained from $\langle \texttt{Dxp} \rangle$ by forgetting open expressions. The class has only one creation method `dxp/dxp`.

$$\frac{M : \langle \texttt{Dxp} \rangle}{(\texttt{dxp}\ \ M) : \langle \texttt{dxp} \rangle}\ \texttt{dxp}$$

where the method may be applied only when $M : \langle \texttt{Dxp} \rangle$ is *closed*.

$\langle$dxp$\rangle$ and $\langle$Dxp$\rangle$

We see, by the construction, that the mother class $\langle\texttt{dxp}\rangle$ is isomorphic to the subclass $\langle\texttt{Dxp0}\rangle$ of $\langle\texttt{Dxp}\rangle$.

Here, we have the following two bijections. They are inverses of the others.

$$\texttt{dxp/dxp} : \langle\texttt{Dxp0}\rangle \to \langle\texttt{dxp}\rangle$$
$$\texttt{dxp/2Dxp} : \langle\texttt{dxp}\rangle \to \langle\texttt{Dxp0}\rangle$$

```
(defun dxp/2Dxp (M)
  (case M
    ((dxp M) M)))
```

The basic functions on $\langle \texttt{dxp} \rangle$

We can define the following three basic functions on $\langle \texttt{dxp} \rangle$.

$$\texttt{dxp/var} : \langle \texttt{Nat} \rangle \to \langle \texttt{dxp} \rangle$$
$$\texttt{dxp/app} : \langle \texttt{dxp} \rangle\ \langle \texttt{dxp} \rangle \to \langle \texttt{dxp} \rangle$$
$$\texttt{dxp/lam} : \langle \texttt{Nat} \rangle\ \langle \texttt{dxp} \rangle \to \langle \texttt{dxp} \rangle$$

```
(defun dxp/var (x)
  (dxp/dxp (Dxp/var x)))

(defun dxp/app (M N)
  (dxp/dxp (Dxp/app (dxp/2Dxp M) (dxp/2Dxp N))))

(defun dxp/lam (x M)
  (dxp/dxp (Dxp/lam (Dxp/close x (dxp/2Dxp M)))))
```

The basic functions on $\langle \texttt{dxp} \rangle$ (cont.)

$$\texttt{dxp/var} : \langle \texttt{Nat} \rangle \to \langle \texttt{dxp} \rangle$$
$$\texttt{dxp/app} : \langle \texttt{dxp} \rangle \; \langle \texttt{dxp} \rangle \to \langle \texttt{dxp} \rangle$$
$$\texttt{dxp/lam} : \langle \texttt{Nat} \rangle \; \langle \texttt{dxp} \rangle \to \langle \texttt{dxp} \rangle$$

By these functions, $\langle \texttt{dxp} \rangle$ becomes an algebraic structure with these algebraic operations. We can also define appropriate recognizers:

`dxp/var?`, `dxp/app?`, `dxp/lam?`

and selectors:

`dxp/var1`, `dxp/app1`, `dxp/app2`, `dxp/lam1`, `dxp/lam2`

for the data structure.

The structure of $\langle \texttt{dxp} \rangle$ makes $\langle \texttt{dxp} \rangle$ an adequate model of the lambda terms.

This can be seen in the next slide.

Translation from ⟨Txp⟩ to ⟨dxp⟩

[see lecture2.pdf]
We have the *homomorphism*:

$$\texttt{Txp/2dxp} : \langle \texttt{Txp} \rangle \rightarrow \langle \texttt{dxp} \rangle$$

```
(defun Txp/2dxp (M)
  (case M
    ((var x)   (dxp/var x))
    ((app M N) (dxp/app (Txp/2dxp M) (Txp/2dxp N)))
    ((lam x M) (dxp/lam x (Txp/2dxp M)))))
```

Translation from $\langle \texttt{Txp} \rangle$ to $\langle \texttt{dxp} \rangle$

[see lecture2.pdf]

We have the *homomorphism*:

$$\texttt{Txp/2dxp} : \langle \texttt{Txp} \rangle \to \langle \texttt{dxp} \rangle$$

```
(defun Txp/2dxp (M)
  (case M
    ((var x)   (dxp/var x))
    ((app M N) (dxp/app (Txp/2dxp M) (Txp/2dxp N)))
    ((lam x M) (dxp/lam x (Txp/2dxp M)))))
```

From this, we have the *isomorphism*:

$$\texttt{Txp/2dxp} : \langle \texttt{Txp} \rangle / {=_\alpha} \to \langle \texttt{dxp} \rangle$$

Alpha equivalence, revisited

Recall that we defined an equivariant function:

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

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, revisited (cont.)

We can now replace this function by the following:

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

```
(defun Txp/=? (M N)
  (=? (Txp/2dxp M) (Txp/2dxp N)))
```

Note that no *renaming* of variables are necessary in the new definition of alpha equivalence.
One can even bypass the old definition and can use the new definition.

β-conversion on $\langle \mathtt{dxp} \rangle$

The following function computes the β-conversion on $\langle \mathtt{dxp} \rangle$.

$$\mathtt{dxp/beta} : \langle \mathtt{dxp} \rangle\ \langle \mathtt{dxp} \rangle \to \langle \mathtt{dxp} \rangle$$

```
(defun dxp/beta (M N)
  (case M
    ((dxp M)
     (case N
       ((dxp N)
        (case M
          ((lam M) (dxp/dxp (Dxp/open M N))))))))
```

β-conversion on $\langle\texttt{dxp}\rangle$ (cont.)

$$\texttt{Dxp/open} : \langle\texttt{Dxp}\rangle\ \langle\texttt{Dxp}\rangle \to \langle\texttt{Dxp}\rangle$$
$$\texttt{Dxp/Open} : \langle\texttt{Dxp}\rangle\ \langle\texttt{Nat}\rangle\ \langle\texttt{Dxp}\rangle \to \langle\texttt{Dxp}\rangle$$

are defined as follows.

```
(defun Dxp/open (M N)
  (Dxp/Open M 0 N))

(defun Dxp/Open (M i N)
  (case M
    ((var x) M)
    ((idx j) (if (=? i j) N M))
    ((app M1 M2)
     (Dxp/app (Dxp/Open M1 i N) (Dxp/Open M2 i N)))
    ((lam M) (Dxp/lam (Dxp/Open M (1+ i) N)))))
```

The class ⟨`dxp`⟩ is not abstract

The basic funtions on ⟨`dxp`⟩, namely, the *constructors*: `dxp/var`, `dxp/app`, `dxp/lam`, and associated recognizers and selectors are *concretely* defined by the users.

This means that the class ⟨`dxp`⟩ is not an abstract data type.