# ソフトウェア論理
# Logic in Computer Software

Yukiyoshi Kameyama

Department of Computer Science, University of Tsukuba

Week 1: Functional Languages and Types





## "Canonical" Example: Power Function

In OCaml:

```
let rec power x n =
  if n = 1 then x
  else  x * (power x (n-1))
```

power 10 1 ⇝ 10

power 10 2 ⇝ 100

power 10 3 ⇝ 1000





## "Canonical" Example: Power Function

In Scheme:

```
(define (power x n)
  (if (= n 1) x
     (* x (power x (- n 1)))))
```

(power 10 1) ⇝ 10

(power 10 2) ⇝ 100

(power 10 3) ⇝ 1000





## "Canonical" Example: Generating Power Function

In Scheme:

```
(define (genpower y n)
  (if (= n 1) y
     (list '* y (genpower y (- n 1)))))
```

(genpower 'x 3) ⇝ (* x (* x x))

Better solution using Quasi-Quotation:

```
(define (genpower y n)
  (if (= n 1) y
     `(* ,y ,(genpower y (- n 1)))))
```

(genpower 'x 3) ⇝ (* x (* x x))

## "Canonical" Example: Generating Power Function

Want to generate more efficient code:

```
(define (double u)
    `(* ,u ,u))
(define (genpower2 y n)
  (if (= n 1) y
    (if (even? n)
        (double (genpower2 y (/ n 2)))
     `(* ,y ,(genpower2 y (- n 1))))))
```

(where "/" should be "quotient" function in Scheme.)

(double 'x) $\rightsquigarrow$ (* x x)

(genpower2 'x 4) $\rightsquigarrow$ (* (* x x) (* x x))





## "Canonical" Example: Generating Power Function

Want to generate more efficient code (2nd try):

```
(define (double u)
    `(let ((z ,u)) (* z z)))
(define (genpower2 y n)
  (if (= n 1) y
    (if (even? n)
        (double (genpower2 y (/ n 2)))
     `(* ,y ,(genpower2 y (- n 1))))))
```

(double 'x) $\rightsquigarrow$ (let ((z u)) (* z z))

(genpower2 'x 4) $\rightsquigarrow$ (let ((z (let ((z x)) (* z z)))) (* z z))





## "Canonical" Example: Generating Power Function

Sometimes we suffer:

```
(define (double u)
    `(let ((z ,u)) (* ,u z)))
(define (genpower3 y n)
  (if (= n 1) y
    (if (even? n)
        (double ,(genpower3 y (/ n 2)))
     `(* ,y ,(genpower3 y (- n 1))))))
```

(genpower3 'x 4) $\rightsquigarrow$ (let ((z (let ((z x)) (* x z)))) (* (let ((z x)) (* x z)) z)) (good)

(genpower3 'z 4) $\rightsquigarrow$ (let ((z (let ((z z)) (* z z)))) (* (let ((z z)) (* z z)) z)) (bad)





## Program Generation

Writing program generators is probably better than hand-writing each code.
But we need to answer questions such as:

- Is it possible to write generators for all cases ?
- Is it easy to write generators for all cases ?
- Is the generated code efficient (compared with hand-written code) ?
- Is the generated code safe and correct ?

PL (programming language) research should answer these questions.

## PL research for Program Generation

In the second half of this course, I will talk about:

- Programming-language support for program generation.
- In particular, a sophisticated type system which:
  - makes it easy (sometimes automatic) to write generators, and gives us safety assurance of generated code.





## Course schedule

- Five weeks (Oct. 7, 14, 21, 28, Nov. 6): Mizutani-sensei
- Nov. 11: Functional Programming and Type System
- Nov. 18: Basics of Program Generation
- Nov. 25: Examples of Program Generation
- Dec. 02: Type system of Program Generation
- Dec. 09: Application of Program Generation
- (no final examination)

Our assumption: we should use a functional programming language when talking about program generation.





We will introduce a small functional programming language, which

- is very small (its syntax is small),
- is statically typed, and
- has clear semantics.





## Expression (式) in Backus Normal Form

$e$ stands for Expression:

| | |
|---|---|
| $e ::= 0 \mid 1 \mid 2 \mid \ldots$ | `int`literal |
| $\mid$ `true` $\mid$ `false` | `bool`literal |
| $\mid x$ | variable |
| $\mid e + e \mid e - e \mid e = e \mid \cdots$ | arithmetic |
| $\mid$ `if` $e$ `then` $e$ `else` $e$ | conditional |
| $\mid \lambda x.\ e \mid e\ e$ | unary function |
| $\mid \lambda(\overline{x}).\ e \mid e\ (\overline{e})$ | multi-ary function |
| $\mid$ `fix` $f(\overline{x}).\ e$ | recursion |
| $\mid (e)$ | |

$\overline{e}$ is a finite sequence of e's.

## Example of expressions

$1 + x$

$\texttt{if}\ x = 0\ \texttt{then}\ y + 1\ \texttt{else}\ f(2 + z)$

$f(g(h(0)))$

$f0$

$\lambda x.\ x + 1$

$(\lambda x.\ x + 1)\ 3$

$(\lambda f.\ f(f\ 3))(\lambda x.\ x + 1)$

$\lambda x.\ \ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ f(x - 1)$

$\texttt{fix}\ f(x).\ \ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ f(x - 1)$





## Lambda Expression (ラムダ式)

$\lambda x.\ e$

represents the function $f$ defined by $f(x) = e$. (but we don't use names such as $f$).

its input is $x$, and output is $e$.

to use the function, we write, e.g., $(\lambda x.\ e)(103)$. (function application)

parenthesis may be omitted, e.g., $(\lambda x.\ e)\ 103$.

functions as first-class data: $(\lambda f.\ f(3) + 5)(\lambda x.\ x + 1)$ (higher-order function)

$\lambda(\overline{x}).\ e$

a function which has multiple arguments.

e.g., $\lambda(x, y).\ x * 3 + y$





## Recursive Function (再帰関数)

$\texttt{fix}\ f(x).\ e$

represent the recursive function $f$ defined by $f(x) = e$.

where $e$ may contain $x$ and $f$ itself (recursive call).

in OCaml, we write `let rec f x = e`.

fix means "fixed point" or "fixpoint" (不動点)





## Programming Example (Sum of 1..x)

$$\texttt{fun1} \stackrel{def}{=} \texttt{fix}\ f(x).\ \ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ (x + (f\ (x - 1)))$$

$$\begin{aligned}
\texttt{fun1}(3) &\rightsquigarrow \texttt{if}\ 3 = 0\ \texttt{then}\ 0\ \texttt{else}\ (3 + (\texttt{fun1}\ (3 - 1))) \\
&\rightsquigarrow \texttt{if}\ \texttt{false}\ \texttt{then}\ 0\ \texttt{else}\ (3 + (\texttt{fun1}\ (3 - 1))) \\
&\rightsquigarrow 3 + (\texttt{fun1}\ (3 - 1)) \\
&\rightsquigarrow 3 + (\texttt{fun1}\ (2)) \\
&\rightsquigarrow 3 + (\texttt{if}\ 2 = 0\ \texttt{then}\ 0\ \texttt{else}\ 2 + (\texttt{fun1}\ (2 - 1))) \\
&\rightsquigarrow^* 3 + (2 + (1 + 0)) \\
&\rightsquigarrow^* 6
\end{aligned}$$

## Programming Example (Product of 1..x)

$$\texttt{fun2} \stackrel{def}{=} \texttt{fix}\ f(x).\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f\ (x-1))$$

$$\begin{aligned}
\texttt{fun2}(3) &\rightsquigarrow \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{fun2}\ (3-1)) \\
&\rightsquigarrow \texttt{if}\ \texttt{false}\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{fun2}\ (3-1)) \\
&\rightsquigarrow 3 * (\texttt{fun2}\ (3-1)) \\
&\rightsquigarrow 3 * (\texttt{fun2}\ (2)) \\
&\rightsquigarrow 3 * (\texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fun2}\ (2-1))) \\
&\rightsquigarrow^* 3 * (2 * (1 * 1)) \\
&\rightsquigarrow^* 6
\end{aligned}$$





## Programming Example (Multiplication)

$$\texttt{fun3} \stackrel{def}{=} \texttt{fix}\ f(x,y).\ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ y + f\ (x-1, y)$$

$$\begin{aligned}
\texttt{fun3}(3,2) &\rightsquigarrow \texttt{if}\ 3 = 0\ \texttt{then}\ 0\ \texttt{else}\ 2 + \texttt{fun3}\ (3-1, 2) \\
&\rightsquigarrow 2 + \texttt{fun3}\ (3-1, 2) \\
&\rightsquigarrow 2 + \texttt{fun3}\ (2, 2) \\
&\rightsquigarrow^* 2 + (2 + \texttt{fun3}\ (1, 2)) \\
&\rightsquigarrow^* 2 + (2 + (2 + \texttt{fun3}\ (0, 2))) \\
&\rightsquigarrow^* 2 + (2 + (2 + 0)) \\
&\rightsquigarrow^* 6
\end{aligned}$$





## Programming Example (Non-termination)

$$\texttt{fun4} \stackrel{def}{=} \texttt{fix}\ f(x).\ f(x)$$

$$\begin{aligned}
\texttt{fun4}\ (0) &\rightsquigarrow \texttt{fun4}(0) \\
&\rightsquigarrow \cdots
\end{aligned}$$





## Syntax vs Semantics (構文と意味)

Syntax determines if this string is a program or not.

Semantics determines what is the result of computing a program.

Cf. in natural languages, syntax defines what are sentences, and semantics defines their meaning. (???)

## Three styles of program semantics

Operational Semantics (操作的意味論)

Specify what are the value(s) of a given program $e$ (or if it does not have a value).

Axiomatic Semantics (公理的意味論)

Specify which properties hold/do not hold for $e$.

Denotational Semantics (表示的意味論、あるいは、外延的意味論)

Specify which partial function (or mathematical object) a given program $e$ corresponds to.





## Example of computation (cont'd)

Let

$$\texttt{double} = \lambda f.\ \lambda x.\ f(f\ x)$$
$$\texttt{inc} = \lambda y.\ y + 1$$

then we compute $(\texttt{double}\ \texttt{inc})\ 1$

$$\begin{aligned}(\texttt{double}\ \texttt{inc})\ 10 &\rightsquigarrow (\lambda x.\ (\texttt{inc}(\texttt{inc}\ x)))\ 10\\ &\rightsquigarrow \texttt{inc}(\texttt{inc}\ 10)\\ &\rightsquigarrow \texttt{inc}(10+1)\\ &\rightsquigarrow (10+1)+1\\ &\rightsquigarrow^* 12\end{aligned}$$





## Example of computation (cont'd)

$$\texttt{gt} \stackrel{def}{=} \texttt{fix}\ f(x,y).\ \texttt{if}\ x = 0\ \texttt{then}\ \texttt{false}\ \texttt{else}$$
$$\texttt{if}\ y = 0\ \texttt{then}\ \texttt{true}\ \texttt{else}\ f\ (x-1, y-1)$$

$$\begin{aligned}&\texttt{gt}\ (2,\ 3)\\ &\rightsquigarrow^* \texttt{gt}\ (1,\ 2)\\ &\rightsquigarrow^* \texttt{gt}\ (0,\ 1)\\ &\rightsquigarrow^* \texttt{false}\end{aligned}$$

$$\begin{aligned}&\texttt{gt}\ (3,\ 2)\\ &\rightsquigarrow^* \texttt{gt}\ (2,\ 1)\\ &\rightsquigarrow^* \texttt{gt}\ (1,\ 0)\\ &\rightsquigarrow^* \texttt{true}\end{aligned}$$





## Quiz

Q1. What does the following expression (function) compute ?
$\texttt{fix}\ f(x).\ \texttt{if}\ \texttt{gt}(x, 100)\ \texttt{then}\ x - 10\ \texttt{else}\ f\ (f\ (x+11))$

## Type System (型システム)

Observation: the syntax is too coarse; we want to exclude BAD programs, for example:

Non-function in the function position.

$$(10\ \texttt{true}) + (\texttt{false}\ 30)$$

Testing non-boolean.

$$\texttt{if}\ 100\ \texttt{then}\ 20\ \texttt{else}\ 30$$

Question: how to detect such BAD programs before executing them (i.e. at the compile time) ?





## Static Type System

To distinguish good programs from bad programs.

If a program $e$ is well typed, the execution of $e$ never goes wrong. ("wrong" means a run-time mismatch of data, e.g., adding boolean and string. Similar to "Segmentation Fault" error in C.)

Other merits.

Types improve readability, maintainability, modularity.

Compilers can make use of type information to generate efficient codes.

Types provide a means of abstraction.





## Assigning Types to Programs

Examples:

$10 + 20 : \texttt{int}$

$\texttt{if true then } 10 \texttt{ else } 20 : \texttt{int}$

$\texttt{if true then } 10 \texttt{ else } \textit{true}$ : Ill typed

$\lambda x.\ \texttt{if true then } x + 10 \texttt{ else } 20 : \texttt{int} \to \texttt{int}$

$\lambda(x, y).\ \texttt{if } y \texttt{ then } x + 10 \texttt{ else } 20 : (\texttt{int}, \texttt{bool}) \to \texttt{int}$

$\lambda y.\ \lambda x.\ \texttt{if } y \texttt{ then } x + 10 \texttt{ else } 20 : \texttt{bool} \to (\texttt{int} \to \texttt{int})$

From now on, we will use Church-style presentation:

$\lambda(x : \texttt{int}).\ \texttt{if true then } x + 10 \texttt{ else } 20 : \texttt{int} \to \texttt{int}$

$\lambda(x : \texttt{int}, y : \texttt{bool}).\ \texttt{if } y \texttt{ then } x + 10 \texttt{ else } 20 : (\texttt{int}, \texttt{bool}) \to \texttt{int}$





## Assigning Types to Programs

More Examples:

$\texttt{if } y \texttt{ then } x + 10 \texttt{ else } 20 : \texttt{int}$

$\texttt{if } y = 30 \texttt{ then } x + 10 \texttt{ else } 20 : \texttt{int}$

$(\texttt{if } y \texttt{ then } x + 10 \texttt{ else } 20) + (\texttt{if } y = 30 \texttt{ then } x + 10 \texttt{ else } 20) : \texttt{int}$?

We need to "remember" the type of free variables, too.

## Assigning Types to Programs

Types of free variables and a program:

$$x : \texttt{int}, y : \texttt{bool} \vdash \texttt{if } y \texttt{ then } x + 10 \texttt{ else } 20 : \texttt{int}$$
$$x : \texttt{int}, y : \texttt{int} \vdash \texttt{if } y = 30 \texttt{ then } x + 10 \texttt{ else } 20 : \texttt{int}$$

Typing Judgment:

$$\Gamma \vdash e : \tau$$

where 、 $\Gamma$ is $x_1 : \sigma_1, \cdots, x_n : \sigma_n$, $e$ is a term (an expression), and $\tau$ is a type.





## The Set of Types

The set of types depend on programming languages. In this language, we have:

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid \sigma \to \tau(\sigma_1, \cdots, \sigma_n) \to \tau$$

Example of types:

$\texttt{int}$

$\texttt{bool}$

$\texttt{int} \to \texttt{int}$

$\texttt{int} \to (\texttt{bool} \to \texttt{int})$

$(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$





## Type System-1

$$\frac{}{\Gamma \vdash x : \sigma}\ \text{var},((x : \sigma) \in \Gamma)$$

$$\frac{}{\Gamma \vdash m : \texttt{int}}\ \text{const1},(m \text{ is an integer literal})$$

$$\frac{}{\Gamma \vdash b : \texttt{bool}}\ \text{const2},(b \text{ is } \texttt{true} \text{ or } \texttt{false})$$





## Type System-2

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \text{plus}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \sigma \quad \Gamma \vdash e_3 : \sigma}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \sigma}\ \text{if}$$

## Type System-3

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda(x : \sigma).\ e : \sigma \to \tau}\ \text{fun}$$

$$\frac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash f : \sigma}{\Gamma \vdash e\ f : \tau}\ \text{apply}$$





## Type System-4

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash e : \tau}{\Gamma \vdash (\texttt{fix}\ (f : \sigma \to \tau)(x : \sigma).\ e) : \sigma \to \tau}\ \text{fix}$$

Note. $\texttt{fix}(f)(x).\ e$ is the function such that $f(x) = e$.





## Example of Type Derivation

$$\frac{\dfrac{}{\cdot \vdash 0 : \texttt{int}} \quad \dfrac{}{\cdot \vdash 1 : \texttt{int}}}{\cdot \vdash 0 + 1 : \texttt{int}}$$





## Example of Type Derivation

Let $\Gamma = x : \texttt{int}, y : \texttt{bool}$.

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash y : \texttt{bool}} \quad \dfrac{}{\Gamma \vdash x : \texttt{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}} \quad \dfrac{}{\Gamma \vdash 1 : \texttt{int}}}{\Gamma \vdash x + 1 : \texttt{int}}}{\Gamma \vdash \texttt{if}\ y\ \texttt{then}\ x\ \texttt{else}\ x + 1 : \texttt{int}}}{y : \texttt{bool} \vdash \lambda(x : \texttt{int}).\ \texttt{if}\ y\ \texttt{then}\ x\ \texttt{else}\ x + 1 : \texttt{int} \to \texttt{int}}}{\cdot \vdash \lambda(y : \texttt{bool}).\ \lambda(x : \texttt{int}).\ \texttt{if}\ y\ \texttt{then}\ x\ \texttt{else}\ x + 1 : \texttt{bool} \to (\texttt{int} \to \texttt{int})}$$

## Example of Type Derivation

Let $\mathtt{BB} = \mathtt{bool} \to \mathtt{bool}$ and $\Delta = x : BB, y : \mathtt{bool}$,

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash x : \mathtt{BB}} \quad \dfrac{}{\Delta \vdash y : \mathtt{bool}}}{\Delta \vdash xy : \mathtt{bool}}}{\Delta \vdash x(xy) : \mathtt{bool}}}{\dfrac{}{x : \mathtt{BB} \vdash \lambda(y : \mathtt{bool}).\ x(xy) : \mathtt{BB}}}{\cdot \vdash \lambda(x : \mathtt{BB}).\ \lambda(y : \mathtt{bool}).\ x(xy)\ :\ \mathtt{BB} \to \mathtt{BB}}$$





## Type System-5

$$\dfrac{\Gamma, x_1 : \sigma_1, \cdots, x_n : \sigma_n \vdash e : \tau}{\Gamma \vdash \lambda(x_1 : \sigma_1, \cdots, x_n : \sigma_n).\ e : (\sigma_1, \ldots, \sigma_n) \to \tau}\ \text{fun'}$$

$$\dfrac{\Gamma \vdash e : (\sigma_1, \ldots, \sigma_n) \to \tau \quad \Gamma \vdash e_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash e_n : \sigma_n}{\Gamma \vdash e\ (e_1, \ldots, e_n) : \tau}\ \text{app'}$$





## Type System-6

$$\dfrac{\Gamma,\ f : (\ldots, \sigma_i, \ldots) \to \tau,\ \ldots, x_i : \sigma_i, \ldots \vdash e : \tau}{\Gamma \vdash \mathtt{fix}\ (f : (\ldots, \sigma_i, \ldots) \to \tau)(\ldots, x_i : \sigma_i, \ldots).\ e\ :\ (\ldots, \sigma_i, \ldots) \to \tau}\ \text{fix}$$

Note. $\mathtt{fix}\ (f : \sigma)(\ldots, x_i : \tau_i, \ldots).\ e$ is a function such that $f(\ldots, x_i, \ldots) = e$.





## Exercises

Define the typing rules for $e - f$ and $e = f$ appropriately. (For $e = f$, you have many choices.)

Write a type derivation for
$x : \mathtt{int} \to \mathtt{int}, y : \mathtt{int} \vdash x(x(x\ y)) : \mathtt{int}$.

Find a term $e$ such that the following judgment is derivable for *any* $\sigma$,$\tau$ and $\rho$:

$$\vdash e\ :\ (\sigma \to \tau) \to ((\tau \to \rho) \to (\sigma \to \rho))$$

Write a type derivation for the following function:

$\mathtt{fix}\ (f : \sigma \to \tau)(x : \sigma).\ \mathtt{if}\ \mathtt{gt}(x, 100)\ \mathtt{then}\ x - 10\ \mathtt{else}\ f(f(x + 11))$

(optional) Is there any type derivation for $x : \sigma \vdash x\ x : \tau$ for some $\sigma$ and $\tau$ ?

## Type Soundness (型システムの健全性)

Strong type soundness means subject reduction and progress.

**Theorem (Subject Reduction)**

*If $\Gamma \vdash e : \sigma$ is derivable, and $e \rightsquigarrow^* e'$, then $\Gamma \vdash e' : \sigma$ is derivable, where $\rightsquigarrow^*$ means computation.*

**Theorem (Progress)**

*If $\Gamma \vdash e : \sigma$ is derivable, and $e$ is not (yet) a value, then there exists a $e'$ such that $e \rightsquigarrow e'$.*





## Benefits of Type System

If strong type soundness holds, then well-typed programs do not go wrong [Hindley and Milner].

> Once a program typechecks (before execution), then
>
> it does not cause a type error during its computation, and
>
> it does not get stuck.

Note. Type soundness does not guarantee the program will terminate, or the program does not cause a run-time error such as "division by zero".





## Type Checking and Type Inference

Type checking (e.g. in C and Java):

> Input: A typing context $\Gamma$, a term $e$, and a type $\tau$.
>
> Output: Yes or No ($\Gamma \vdash e : \tau$ is derivable or not)

Type inference (e.g. in ML family and Haskell):

> Input: $e$
>
> Output:
>
> 1: $\Gamma$ and $\tau$ (such that $\Gamma \vdash e : \tau$ is derivable)
> 2: Fail
>
> Question: but how to output infinitely many answers?





## Principal Typing

Problem: Infer the type of $\lambda(.\, x : \sigma).x$.
Solution: $\texttt{int} \to \texttt{int}$, $\texttt{bool} \to \texttt{bool}$, $(\texttt{int} \to \texttt{bool}) \to (\texttt{int} \to \texttt{bool})$, $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}), \cdots$

We must represent an infinitely many solutions in a finitistic way.

> A typing of a term $e$ is $(\Gamma, \sigma)$ such that $\Gamma \vdash e : \sigma$ is derivable.
>
> Principal typing of $e$ is a typing of $e$ such that any other typing of $e$ can be obtained by it (using substitution and weakening).

**Theorem (Principal Typing)**

*If $e$ is typable, its principal typing exists.*

## Summary

Types play an essential role in most modern programming languages.

- Type system guarantees a certain static property,
- by ruling out syntactically bad (ill-typed) programs.
- The "type soundness" property is the key.

Type systems in programming languages.

- Core: simply typed lambda calculus (as in this lecture).
- Widely used: record, variant, polymorphic type, types for objects
- Research: dependent types, higher-order types





## Exercise

1. Derive the type for factorial function.
2. Does $\lambda x.\ x\ x$ have a type ?