Principles of Programming Languages

I.2 Lambda Calculus

Dr. Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz

LAMBDA CALCULUS

developed by Alonzo Church, 1930s

Formal theory for computable functions

- universal model for computations ➔ Turing complete

Consists of

- Syntax in the form of **lambda expressions**
- Operational semantics in the form of **conversion rules**

Used for

- reasoning about computable functions
- formal definition of semantics of programming languages
- model for implementation of functional programming languages
  - functional languages are direct implementations of lambda calculus
  - lambda calculus is basis for the execution model of functional languages,

JKU

Lambda Calculus

- Syntax
- Conversion rules
- Evaluation strategies
- Summary

Syntax of Lambda Expressions

Lambda expressions

```
Lamba-expr =
      Variable
    | λ Variable {Variable}  . Lambda-expr
    | Lambda-expr Lambda-expr {Lambda-expr} .
```

lambda abstraction

also named ***lambda function***

function application

```
    | Constant
```

names for lambda functions (optional)

LAMBDA ABSTRACTION

Function definitions by lambda abstractions

- Lambda abstraction = anonymous function definition

arguments

body

```
λ Variable {Variable} . Lambda-expr
```

bound variables

defining expression
*containing **bound** and **free** variables*

<u>Examples</u>:

two arguments

```
λ x  .  f x
```

```
λ x y . f x y
```

function of x *is defined by* *lambda expression*

FUNCTION APPLICATION

Prefix notation of function applications

- juxtaposition of function and argument(s)

function

argument

poosibly more arguments

```
Lambda-expr Lambda-expr {Lambda-expr}
```

lambda expression defining function

lambda expressions defining arguments

Examples:

```
f a
```

```
f a b
```

$$\underbrace{\underbrace{(\lambda\ x\ .\ f\ x\ y)}_{\text{lambda abstraction}}\ a}_{\text{function application}}$$

$$\underbrace{\underbrace{(\lambda\ x\ y\ .\ f\ x\ y)}_{\text{lambda abstraction}}\ a\ b}_{\text{function application}}$$

Bound and Free Variables

- A variable is ***bound*** if there is an enclosing lambda abstraction which binds it; otherwise the variable is called ***free*** in the lambda abstraction

```
λ y . f x y
```

y is ***bound***, **f** and **x** are ***free*** in expression **f x y**

- Bound variables get their values by arguments of a function application

```
(λ y . f x y) b
```

→

```
f x b
```

- free variables may be bound by some surrounding lambda abstraction, e.g.,

```
λ x . (λ y . f x y)
```

```
(λ x . (λ y . f x y)) a b
```

→

```
f a b
```

now **x** is **bound** by **outer lambda abstraction** (f still free)

Multiple Arguments and Curry-Form

after Haskell B. Curry,
US-American logician, 1900-1982.

Lambda abstraction with **multiple bound variables**

```
λ x y . f x y
```

two variables **x** and **y** are **bound**

Curry-form: lambda abstractions always with a **single variable**

- expression can be a further lamdba abstraction with next variable bound

```
λ x . (λ y . f x y)
```

first abstraction **binds x** and
is defined by **second** abstraction which **binds y**

Currying

- **Building Curry-form** by successively forming **one argument abstractions**

```
λ x y z  .  f x y z   ➔   λ x . (λ y . (λ z . f x y z ))
```

- both forms are **equivalent**!

Constants

- In (pure) lambda calculus no functions and literals exist per se
  - they all are finally expressed by lambda abstractions

- We informally introduce **bindings** of **lambda abstractions** to **names**, like

```
TRUE = λ t f . t
FALSE = λ t f . f
```

```
AND = λ a b . a  b (λ t f . f)
…
```

```
0 = λ f. λ x. x
1 = λ f. λ x. f x
…
```

```
+ = λm.λn.λf.λx. m f (n f x)
* = …
```

```
DOUBLE = λ x . + x x
```

or just

```
DOUBLE x = + x x
```

Lambda Calculus

- Syntax
- Conversion rules
- Evaluation strategies
- Summary

Operational Semantics

The operational semantics of lambda calculus is defined by

conversion rules

which specify how to transform one lambda expression into an equivalent lambda expression

Conversions work in both ways:

- ➔ **Reduction**: from more complex expression to simpler expression
- 🡐 **Abstraction**: from simple expression to more complex expression

There are three conversions

- **β-reduction**: Applying function to arguments
- **η-reduction**: Simplifying functions by reducing number of arguments

eta

- **α-conversion**: Renaming of bound variables to avoid name clashes

Abstractions just the opposite

Variable Substitution

A substitution

```
expr [A/x]
```

of a variable **x** in an expression **expr** by a value **A** is defined as follows:

```
v [A/x]           = A                 if v = x
v [A/x]           = v                 if v ≠ x
(λ v . E ) [A/x] = (λ v . E[A/x])    if v ≠ x
(λ v . E ) [A/x] = (λ v . E)         if v = x
(F E) [A/x]       = (F[A/x]  E[A/x])
```

Examples:

```
x[2/x] = 2

y[2/x] = y

(λ y . x y)[2/x] =
(λ y . (x y)[2/x]) =
(λ y . 2 y)

(λ x . x y) [2/x] =
(λ x . x y)

((λ y . x y)  x)[2/x] =
((λ y . x y)[2/x]  x[2/x] ) =
((λ  y . 2 y)  2)
```

New argument x shadows x

Beta-Reduction

Reducible term (redex)

- A reducible term (*redex*) is a function application where left side is a lambda abstraction

```
(λ x . expr) A
```

β-reduction of redex

- replacing **bound variable** in defining **expression** by **argument expression**

$$(\lambda\ x\ .\ expr)\ A \quad \rightarrow_{\beta} \quad expr[A/x]$$

β-abstraction

- β-abstraction is just inverse of β-reduction
  and works by introducing a lambda function with a bound variable

$$(\lambda\ x\ .\ expr)\ A \quad \leftarrow_{\beta} \quad expr[A/x]$$

Examples of Beta-Reductions

```
  (λ x .  + x 1) 4

→         + 4 1
```

```
  (λ f x . f  x)  (λy . * y  y) 3

→ (λ   x . (λy . * y  y) x) 3

→          (λy . * y  y)     3

→                * 3 3

→                9
```

function * with built-in functions with own reduction rule!

Reductions with Functions in Curry-form

Lamda function in Curry form

```
λ x . ( λ y . + x y)
```

<u>Example application</u>:

Partial application with one argument results in lambda function

```
(λ x . (λ y . + x y )) 1
```

$\rightarrow$

```
(λ y . + 1  y )
```

Result of function application
is a **function** ➔ **partial application**

Resulting lambda function can again be applied with next argument

```
(λ y .  +  1  y ) 2
```

which can again be applied

$\rightarrow$

```
+  1  2
```

Alpha-Conversion

- **α-conversion** of λ-abstractions
  - **Renaming** of bound variable in λ-abstraction

Only needed to avoid name clashes of variables!

```
λ x . expr    ↔α[y/x]    λ y . expr[y/x]
```

i.e., all occurrences of variable **x** in **expr** are replaced by variable **y**

Example:

```
               λx . + x x
→α [y/x]       (λx . + x x)[y/x]
→              λy .  + y y
```

Note: We will assume unique variable names and neglect name clashes in the sequel!

ETA-CONVERSION

η-conversion of λ-expressions

$\lambda\ x\ .\ F\ x \quad \leftrightarrow_{\eta} \quad F$

which means that we can **remove the bound variable x** in a lambda abstraction if expression is just application of **F** with **x**

Example:

$\lambda\ x\ .\ (\lambda y\ .\ +\ y\ y)\ x \quad \leftrightarrow_{\eta} \quad (\lambda y\ .\ +\ y\ y)$

Explanation: by applying function

$\lambda\ x\ .\ (\lambda y\ .\ +\ \boxed{y}\ \boxed{y})\ x$

$\rightarrow_{\beta}$ $(\lambda x\ .\ +\ x\ x)$

$\rightarrow_{\alpha\ [y/x]}$ $(\lambda y\ .\ +\ y\ y)$

Evaluation of Lambda-Expressions

Normal form

- A lambda expression is in **normal form**
  iff it **does not contain any reducible term**!

Evaluation by applying reduction rules

- **choose any redex** in the expression
- **reduce the redex** using applicable reduction rules
- until **no redex exists** and the expression is **in normal form**

Encodings of Computation Domains

Lambda calculus for formally defining computation domains

- **lambda abstractions** define **values** and **functions**
- **conversion rules** give **semantics**

➔ Lambda calculus can express **all computable functions** just by **lambda abstractions**

Church Encoding of Boolean Algebra (1/10)

Boolean algebra encoded in lambda calculus

- Boolean values as lambda abstractions with two arguments

True : Projection to the 1st argument t

```
True = (λ t f . t)
```

choose first argument

False: Projection to the 2nd argument f

```
False = (λ t f . f)
```

choose second argument

- Boolean functions as lambda abstractions

AND

False

```
AND = (λ a b . a b (λ t f . f))
```

OR

True

```
OR = (λ a b . a (λ t f . t) b)
```

NOT

False True

```
NOT = λ a . a (λ t f . f) (λ t f . t)
```

IF

```
IF =  (λ c a b . c  a  b)
```

choose from a or b

where **c** has to reduce to a Boolean value

Church Encoding of Boolean Algebra (2/10)

Reduction rule for AND

```
 AND                              True           expr
(λ a b . a b (λ t f . f))   (λ t1 f1 . t1)   expr
```

Reduction:

```
(λ a b . a b (λ t f . f))  (λ t1 f1 . t1)   expr
(λ   b . (λ t1 f1 . t1) b (λ t f . f))   expr
        (λ t1 f1 . t1) expr (λ t f . f)
         (λ   f1 . expr)  (λ t f . f)
                 expr
```

Reduction rule:

```
AND True expr => expr
```

Church Encoding of Boolean Algebra (3/10)

Reduction rule for AND

```
  AND                                  False       expr
(λ a b . a b (λ t f . f) )  (λ t1 f1 . f1)  expr
```

Reduction:

```
(λ a b . a b (λ x y . y)) (λ t1 f1 . f1) expr
(λ   b . (λ t1 f1 . f1) b (λ t f . f)) expr
         (λ t1 f1 . f1) expr (λ t f . f)
           (λ f1 . f1) (λ t f . f)
                 (λ t f . f)
```

False

Reduction rule:

```
AND False expr   => False
```

Church Encoding of Boolean Algebra (4/10)

Reduction rule for OR

```
 OR                               True         expr
(λ a b . a (λ t f . t) b )   (λ t1 f1 . t1)   expr
```

Reduction:

```
(λ a b . a (λ t f . t) b) (λ t1 f1 . t1) expr
(λ   b . (λ t1 f1 . t1) (λ t f . t) b )  expr
        (λ t1 f1 . t1) (λ t f . t) expr
         (λ   f1 . (λ t f . t)) expr
                   (λ t f . t)
```

True

Reduction rule:

```
OR True expr    => True
```

Church Encoding of Boolean Algebra (5/10)

Reduction rule for OR

```
 OR                        False     expr
(λ a b . a (λ t f . t) b ) (λ t1 f1 . f1) expr
```

Reduction:

```
(λ a b . a (λ t f . t) b ) (λ t1 f1 . f1) expr
(λ   b . (λ t1 f1 . f1) (λ t f . t) b ) expr
         (λ t1 f1 . f1) (λ t f . t) expr
          (λ   f1 . f1) expr
                expr
```

Reduction rule:

```
OR False expr    => expr
```

CHURCH ENCODING OF BOOLEAN ALGEBRA (7/10)

Reduction rule for NOT

```
NOT                                 False
λ a. a (λ t f . f) (λ t f . t)     (λ t1 f1 . f1)
```

Reduction:

```
λ a. a (λ t f . f) (λ t f . t)    (λ t1 f1 . f1)
  (λ t1 f1 . f1) (λ t f . f) (λ t f . t)
   (λ   f1 . f1 ) (λ t f . t)
          (λ t f . t)
```

True

Reduction rule:

```
NOT False     => True
```

Church Encoding of Boolean Algebra (6/10)

Reduction rule for NOT

```
 NOT                                     True
λ a. a (λ t f . f) (λ t f . t)   (λ t1 f1 . t1)
```

Reduction:

```
λ a. a (λ t f . f) (λ t f . t)  (λ t1 f1 . t1)
  (λ t1 f1 . t1) (λ t f . f) (λ t f . t)
(λ       f1 . (λ t f . f) ) (λ x y . x)
          (λ t f . f)
```

False

Reduction rule:

```
NOT True      => False
```

JKU

Church Encoding of Boolean Algebra (8/10)

Reduction rule for IF

```
IF                    True   THEN   exprA ELSE exprB
(λ c a b . c a b)  (λ t1 f1 . t1)  exprA      exprB
```

Reduction:

```
(λ c a b . c a b)  (λ t1 f1 . t1)  exprA      exprB
(λ   a b . (λ t1 f1 . t1) a b) exprA      exprB
(λ     b . (λ t1 f1 . t1)  exprA b) exprB
           (λ t1 f1 . t1) exprA exprB
           (λ    f1 . exprA) exprB
                      exprA
```

Reduction rule:

```
IF  True THEN exprA ELSE exprB

  =>

exprA
```

Church Encoding of Boolean Algebra (9/10)

Reduction rule for IF

```
IF                False     THEN exprA ELSE exprB)
(λ c a b . c a b) (λ t1 f1 . f1) exprA     exprB
```

Reduction:

```
(λ c a b . c a b)  (λ t1 f1 . f1)  exprA  exprB
(λ   a b . (λ t1 f1 . f1) a b) exprA  exprB
(λ     b . (λ t1 f1 . f1) exprA b) exprB
           (λ t1 f1 . f1) exprA exprB
           (λ    f1 . f1) exprB
                          exprB
```

Reduction rule:

```
IF  True THEN exprA ELSE exprB
   =>
exprB
```

Church Encoding of Boolean Algebra (10/10)

Formal semantics of Boolean functions by reduction rules:

```
(AND True expr)                      =>   expr
(AND False expr)                     =>   False
(OR True expr)                       =>   True
(OR False expr)                      =>   expr
(NOT True)                           =>   False
(NOT False)                          =>   True
(IF True  THEN exprA ELSE exprB)     =>   exprA
(IF False THEN exprA ELSE exprB)     =>   exprB
```

Church Encoding of Integer Arithmetic (1/2)

Similar encodings exist for other values and functions

- Natural numbers
  - constants

```
0 = λf.λx. x
1 = λf.λx. f x
2 = λf.λx. f (f x)
3 = λf.λx. f (f (f x))
...
n = λf.λx. fⁿ x
```

  - functions

```
SUCC = λn.λf.λx. f (n f x)
```

```
PLUS = λm.λn.λf.λx. m f (n f x)
```

```
MULT = λm.λn.λf. m (n f)
```

```
PRED = λn.λf.λx. n (λg.λh. h (g f))
                   (λu. x) (λu. u)
```

Church Encoding of Integer Arithmetic (2/2)

- Example reduction
  - 1 + 2 = 3

```
PLUS                                   1              2

(λm.λn.λf.λx. m f (n f x)) (λf1.λx1. f1 x1) (λf2.λx2. f2 (f2 x))

(   λn.λf.λx. (λf1.λx1. f1 x1) f (n f x)) (λf2.λx2. f2 (f2 x2))

(      λf.λx. (λf1.λx1. f1 x1) f ((λf2.λx2. f2 (f2 x2) f x)))

(      λf.λx. (λf1.λx1. f1 x1) f ((    λx2. f (f x2) x)))

(      λf.λx. (λf1.λx1. f1 x1) f ( f (f x)))

(      λf.λx. (    λx1. f x1) ( f (f x)))

(      λf.λx. ( f ( f (f x))))

        λf.λx. f (f ( f x))    3
```

Church Encoding of Lists

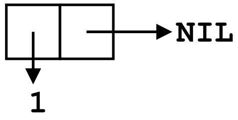
List functions

```
NIL   = λx.λy . y               empty list
CONS  = λx.λy.λz . z x y        pair of two values
HEAD  = λp. p(λx.λy . x)        first of cons pair
TAIL  = λp. p(λx.λy . y)        second of cons pair
```

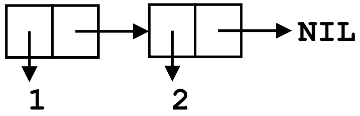
- lists: recursive CONS pairs with **NIL** as last element

NIL

```
NIL
λx.λy . y
```




```
CONS                        1 NIL
(λx1.λy1.λz1 . z1 x1 y1) 1 (λx2.λy2 . y2)
```




```
CONS                     1 ( CONS                     2 NIL          ))
(λx1.λy1.λz1 . z1 x1 y1) 1 ((λx2.λy2.λz2 . z2 x2 y2) 2 (λx3.λy3 . y3)))
```

Church Encoding of Lists

List functions

```
NIL   = λx.λy . y
CONS  = λx.λy.λz . z x y
HEAD  = λp. p(λx.λy . x)
TAIL  = λp. p(λx.λy . y)
```

- NIL: empty list
- CONS: pair of two values
- HEAD: first of cons pair
- TAIL: second of cons pair

■ <u>Applying **HEAD:**</u>

```
HEAD                 (CONS                         1 NIL))
λp. p(λx.λy . x) ((λx1.λy1.λz1 . z1 x1 y1) 1 (λx2.λy2 . y2))
λp. p(λx.λy . x) ((    λy1.λz1 . z1 1  y1)   (λx2.λy2 . y2))
λp. p(λx.λy . x)  (         λz1 . z1 1  (λx2.λy2 . y2))
   (λz1 . z1 1  (λx2.λy2 . y2)) (λx.λy . x)
          (λx.λy . x) 1  (λx2.λy2 . y2))
          (   λy . 1)    (λx2.λy2 . y2))
                1
```

Church Encoding of Lists

List functions

```
NIL   = λx.λy . y              empty list
CONS  = λx.λy.λz . z x y       pair of two values
HEAD  = λp. p(λx.λy . x)       first of cons pair
TAIL  = λp. p(λx.λy . y)       second of cons pair
```

- Applying **TAIL:**

```
TAIL              (CONS                     1  NIL))
λp. p(λx.λy . y) ((λx1.λy1.λz1 . z1 x1 y1) 1 (λx2.λy2 . y2))
λp. p(λx.λy . y) ((    λy1.λz1 . z1 1 y1)    (λx2.λy2 . y2))
λp. p(λx.λy . y)  (        λz1 . z1 1  (λx2.λy2 . y2))
   (λz1 . z1 1  (λx2.λy2 . y2)) (λx.λy . y)
         (λx.λy . y) 1  (λx2.λy2 . y2))
         (   λy . y)    (λx2.λy2 . y2))
             (λx2.λy2 . y2)   nil
```

LAMBDA CALCULUS

- Syntax
- Conversion rules
- Evaluation strategies
- Summary

Example: Different Reduction Orders (1/2)

Reduction rule for AND

```
AND                                True            expr
(λ a b . a b (λ t f . f))   (λ t1 f1 . t1)   expr
```

Reduction:

```
(λ a b . a b (λ t f . f))   (λ t1 f1 . t1)   expr
(λ   b . (λ t1 f1 . t1) b (λ t f . f))   expr
        (λ t1 f1 . t1) expr (λ t f . f)
         (λ   f1 . expr)   (λ t f . f)
                expr
```

Reduction rule:

```
AND True expr    => expr
```

Example: Different Reduction Orders (2/2)

Reduction rule for AND

```
AND                          True            expr
(λ a b . a b (λ t f . f))   (λ t1 f1 . t1)  expr
```

Alternative Sequence of reductions:

```
(λ a b . a b (λ t f . f))  (λ t1 f1 . t1)   expr
(λ   b . (λ t1 f1 . t1) b (λ t f . f))  expr
(λ   b . (λ    f1 . b)    (λ t f . f))  expr
(λ   b .            b)                  expr
                 expr
```

Reduction rule:

```
AND True expr    => expr
```

different reduction order
but same result !

Evaluation of Lambda Expressions

Recall: ***Evaluation*** of lambda expressions by applying reduction rules

- **choose any redex** in the expression
- **reduce the redex** using applicable reduction rules (mainly β-reduction)
- until **no redex** exists and the expression is in **normal form**

does not contain a redex

Questions:

1) **Which redex** should we **choose** and therefore in **which order** apply the reductions?
2) Is the **result (= normal form) independent** of the **chosen order** of reductions?

Answers:

1) **Different strategies** applicable and we distinguish between **strict** and **non-strict** evaluation strategies!
2) **Yes**, results (= normal forms) will be the **same if reached**,
   but it is possible that with **one order the normal form is reached**
   while with **another it is NOT!**

CHURCH-ROSSER THEOREM I

Definitions:

Let $\rightarrow^*$ denote a **series of reductions**

Let $\leftrightarrow^*$ denote a **series of conversions (abstractions and reductions**)

Two expressions $\boldsymbol{E_1}$ **and** $\boldsymbol{E_2}$ **are equivalent** if there is a **conversion** $\boldsymbol{E_1} \leftrightarrow^* \boldsymbol{E_2}$.

Church-Rosser Theorem I:

$$\boldsymbol{E_1 \leftrightarrow^* E_2 \;\; \Rightarrow \; \exists\, E : E_1 \rightarrow^* E \;\; \wedge \;\; E_2 \rightarrow^* E}$$

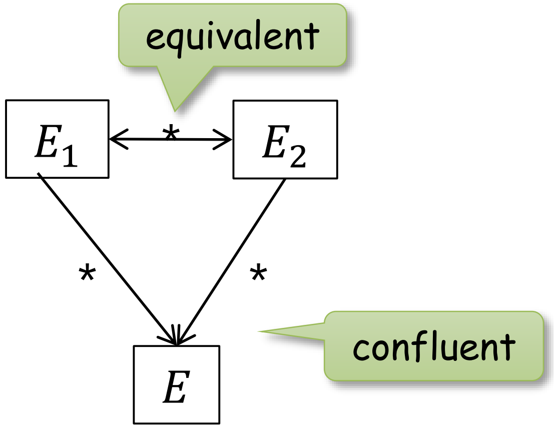
If two expressions $\boldsymbol{E_1}$ and $\boldsymbol{E_2}$ are **equivalent**

$$\boldsymbol{E_1} \leftrightarrow^* \boldsymbol{E_2}$$

then there exists an expression $\boldsymbol{E}$ so that $\boldsymbol{E_1}$ and $\boldsymbol{E_2}$ can be **reduced** to $\boldsymbol{E}$

$$\boldsymbol{E_1} \rightarrow^* \boldsymbol{E} \quad \text{and} \quad \boldsymbol{E_2} \rightarrow^* \boldsymbol{E}$$

That means lambda expressions are **confluent:**



Reduction to Normal Form (1/2)

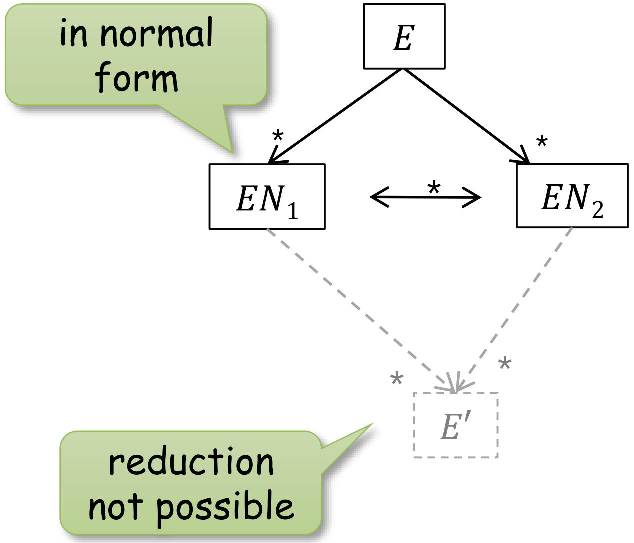
From the Church-Rosser theorem I it directly follows:

Lemma:

No expression can be converted to two distinct normal forms.

Proof sketch: (by establishing a **contradiction** to Church-Rosser theorem I)

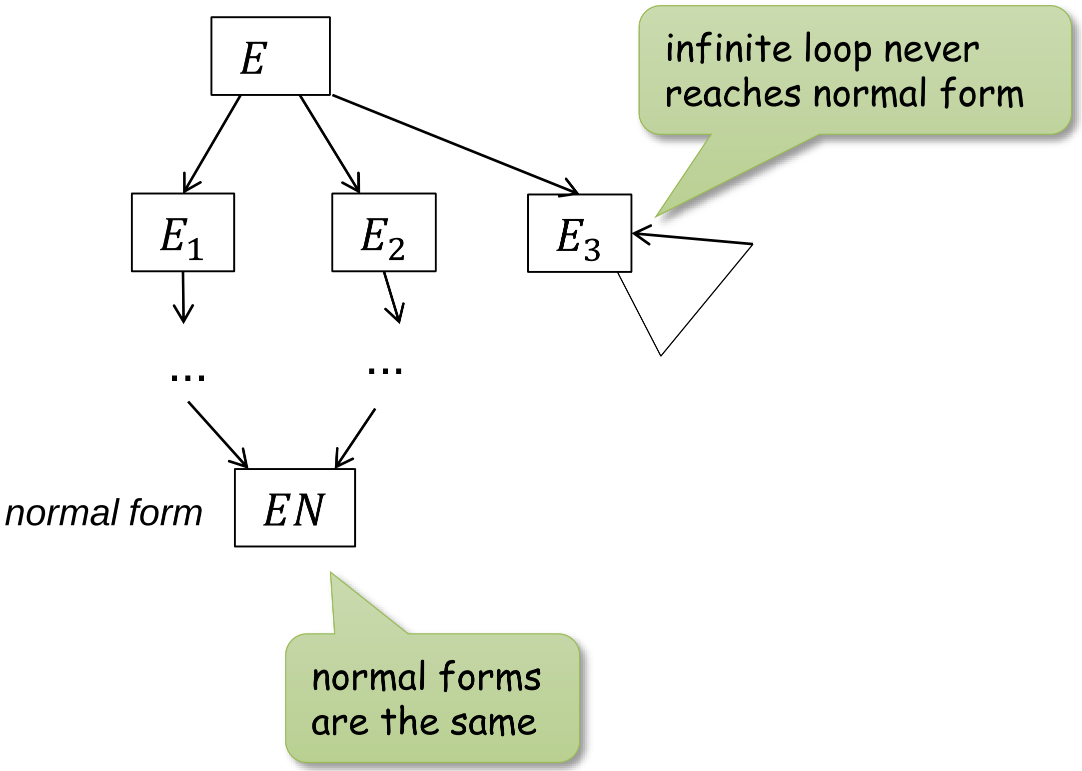
- When it is possible to reduce an expression ***E*** to two distinct expressions ***EN₁*** and ***EN₂*** which both are in normal form, then ***EN₁*** and ***EN₂*** are equivalent $\boldsymbol{EN_1} \leftrightarrow^* \boldsymbol{EN_2}$.
- Then according to Church-Rosser theorem I, there has to be a reduction of ***EN₁*** and ***EN₂*** to a common expression ***E'***.
- However ***EN₁*** and ***EN₂*** are in normal form and cannot be reduced, which represents a contradiction to Church-Rosser theorem I.



Reduction to Normal Form (2/2)

Interpretation of uniqueness of normal form:

- if **two reductions reach normal form**, then they **are the same**
- But there **can be reductions** which run **into an infinite loop** and will **never reach normal form**!



EVALUATION STRATEGIES

Strategies for selecting reducible terms

Strict evaluation (*eager evaluation*, *applicative evaluation*)

- **Call-by-value:**
  actual argument expressions are **evaluated** and **values** replace formal parameters

Non-strict evaluation (*lazy evaluation*)

Algol, Scala !

- **Call-by-name:**
  actual argument expressions are passed **unevaluated** and expressions replace formal parameters

- **Normal-order:**
  Call-by-name with leftmost, outermost redex reduced first

Haskell !

- **Call-by-need:**
  variant of normal-order where expressions are evaluated only when needed and only once!

Strict Evaluation: Call-by-value

Example:

- Example function

```
SQUARE = λ x . * x x
DOUBLE = λ n . * 2 n
```

Strict evaluation: Evaluate argument expressions first

```
SQUARE (DOUBLE 3)
```

argument expression first!

```
    SQUARE (DOUBLE 3)

    (λ x . * x x) ((λ n . * 2 n) 3)
→ (λ x . * x x) (* 2 3)
→ (λ x . * x x) 6
→ * 6 6
→ 36
```

Non-Strict Evaluation: Normal-order

Example:

- Example function

Non-strict evaluation with leftmost, outermost redex reduced first

```
SQUARE (DOBLE 3)
```

left-most, outmost first!

```
    SQUARE           (DOUBLE 3)

    (λ x . * x x) ((λ n . * 2 n) 3)
→   (* ((λ n . * 2 n) 3) ((λ n . * 2 n) 3))
→   (*         (* 2 3)    ((λ n . * 2 n) 3))
→   (*          6         ((λ n . * 2 n) 3))
→   (*          6          (* 2 3) )
→   (*          6           6 )
→    36
```

Assumption:
built-in function * strict

Evaluation Strategies: Examples (1/3)

- Recall: reduction rules for logical operators

```
AND True expr = expr
AND False expr = False
```

```
OR True expr = True
OR False expr = expr
```

Example expression:

```
(AND False (AND True (OR False True))
```

- ☐ Strict evaluation (arguments first):

```
  (AND False (AND True (OR False True)))
→ (AND False (AND True True))
→ (AND False True)
→ False
```

- ☐ Normal-order evaluation (left-most, outer-most first):

```
(AND False (AND True (OR False True)))
→ False
```

Short circuit evaluation!

Evaluation Strategies: Examples (2/3)

- Recall function definition IF:

```
IF True exprA exprB  = exprA
IF False exprA exprB = exprB
```

<u>Example expression:</u>

```
(IF (!= x 0)  (/ a x)  0)
```

- ☐ Strict evaluation:

Assuming x == 0!

```
 (IF (!= x 0)  (/ a x)  0)
→ (IF False (/ a x)  0)
```

Error: division by 0!

- ☐ Non-strict evaluation:

```
   (IF (!= x 0)  (/ a x)  0)
→  (IF False (/ a x)  0)
→  0
```

equivalent to built-in evaluation rule of if-statement in strict languages!

Evaluation Strategies: Examples (3/3)

- Function definitions

```
INFINITE = λ x . INFINITE (+ x 1)
FRIST = λ x y . x
```

Example expression:

```
FRIST 1 (INFINITE 1)
```

- Strict evaluation

evaluate argument first

```
  FRIST 1 (INFINITE 1)
→ FRIST 1 (INFINITE 2)
→ FRIST 1 (INFINITE 3)
→ FRIST 1 (INFINITE 4)
→ ... infinite loop ...
```

- Normal-order evaluation

```
  FRIST 1 (INFINITE 1)
→(λ x y . x) 1 (INFINITE 1)
→(λ   y . 1)   (INFINITE 1)
→ 1
```

argument value not used

Church-Rosser Theorem II

Church-Rosser Theorem II:

If $\boldsymbol{E_1} \rightarrow^* \boldsymbol{E_2}$ (i.e., expression $\boldsymbol{E_1}$ **can be reduced** to expression $\boldsymbol{E_2}$) and $\boldsymbol{E_2}$ is in **normal form**, then there exists a **normal-order reduction** sequence from $\boldsymbol{E_1}$ to $\boldsymbol{E_2}$.

Consequence:

Normal-order reduction will **always find the normal form if such a reduction** exists, while **other reduction** sequences may fail and **run into infinite loops**.

Lambda Calculus

- Syntax
- Conversion rules
- Evaluation strategies
- Summary

JKU

Summary Lambda Calculus

- Lambda-expressions
  - **Variable** symbols **x, y, …**
  - **Function definitions** **λ x . λ–expr**
  - **Function applications** **λ–expr λ–expr**
- Computation by **β-reduction** of lambda-expressions
  - **term replacement** **(λ x . expr) A** $\rightarrow_\beta$ **expr [A/x]**
- Reduction can be done in any order
  - different **evaluation strategies**: **strict** evaluation versus **non-strict** evaluations
  - **normal-order** evaluation is more **„reliable“** as it will result in normal form when possible
- Lambda-expressions can represent any computable function

  ➔ **Turing-complete**
- Theoretical basis for **formal definition of semantics** of programming languages
- **Model for implementation** of functional programming languages
  - **Lisp** is an implementation of lambda calculus with **strict** evaluation semantics
  - **Haskell** is implementation of lambda calculus with **call-by-need** evaluation semantics

What you might be asked in the final exam

- Describe what the lambda calculus is
  - its purpose and its use
- Explain lambda-expressions
  - syntax plus explanation of different terms
- Reductions (β-reduction, α-reduction, η-reduction)
  - how reductions work and for what they are used
- β-reduction of non-trivial lambda-expressions
  - see reduction examples for Booleans and numbers above
- Explain Church-Rosser theorems I and II
  - what they express and what their implications are
- Name, explain and compare the different evaluation strategies