Computer sci

13 - Programs as data and the λ-calculus

Winter 2020/21

Torsten Grust
University of Tübingen, Germany

1 ¦ New language level: *DMdA - advanced*

We switch to the next language level ***The power of Abstraction - advanced*** to. Changes or new:

1. New **output format for lists** (⋯) in the REPL:

```
> '( 1 2 ... □ )
( list '1 '2 ... '□ )

> empty
()

>
```

2. **Polymorphic equality test** equal? for any values:

```
(: equal? ( % a% b -> boolean ))
```


Quoting: Programs *are* data

3. Let ' be any expression. Then (() * +, ') yields the **representation** of ' - ' is *not* evaluated:


```
( odds 42 )                  ⇝ 42         } literals
( quote "Leia" )             ⇝ "Leia"     } represent
( quote #t )                 ⇝ #t         } yourself

( quote ( + 40 2 )) ⇝ ( + 40 2 )                   } Represented
( quote ( lambda ( x ) x )) ⇝ ( lambda ( x ) x )   } as a list
```

Syntactic sugar: ' ' ≡ (quote') .

⇒ Compact notation of *literal* lists (literals 7¡):

```
'( 71 72... 7□ ) ⇝ ( list 71 72... 7□ )
'()              ⇝ empty
```


Symbols: Representation of identifiers / names

What exactly is (first '(* 1 2)) ? What are 9:; <=: , x , + in '(9:; <=: (x) (+ x 1)) ?

New signature symbol to represent **identifiers** (Names) in programs:

efficient internal representation (no duplicates),
efficiently comparable (using equal?),
no access to the individual characters of the symbol.

Operations on symbols:

```
(: symbol?          ( % a -> boolean ))   (symbol? '*) ⇝ #t
(: symbol-> string ( symbol -> string ))   inverse
(: string-> symbol ( string -> symbol ))   functions
```


Page 5

2 ¦ The λ-calculus

The **λ-calculus** is a notation that can *be arbitrary* (for a
Computers at all) can represent calculable functions.

Developed in the 1930s by **Alonzo Church** (* 1903, † 1995) as new foundation of mathematics (but the mathematicians preferred that axiomatic set theory ...). Since in use as a theoretical substructure of programming languages.

Alonzo Churc

> “ *There may, indeed, be other applications of the system [the λ-calculus] than its use as a logic.* ”

Page 6

The syntax of the λ-calculus

The set of **expressions** > (*expressions*) of the λ-calculus is defined recursively (@ : infinite set of variable names):

- ∀ B ∊ @ : B ∊> [D: EF:
- ∀ '1 ∊> , '2 ∊> : ('1 '2) ∊> [HII9FJ: +

KK
Function argument

- ∀ B ∊ @ , '1 ∊> : (λB.'1) ∊> [H <L + E

KK
Parameters trunk

☐ This defines (only) the **syntax of** the λ-calculus. A We need **semantics** or meaning in these expressions still lend.

Page 7

The syntax of the λ-calculus

Examples of syntactically correct expressions of the λ-calculus:

N ∊>	
(λN.N) ∊>	Identity function
(λN.O) ∊>	Function ignores argument N, returns e.g.

((P ") N) ∊> application of P to" and N (currying)
(λP. (P ")) ∊> application of Arg P to" (HOF, type ⓘ)

(λP. (P "))
KK

Variables are either U, <) G =, G / VE, F

Agreed abbreviation in the λ-calculus (currying):

('$_1$' $_2$ '$_3$...' □) ≡ (··· (('$_1$' $_2$) '$_3$) ...' □)
(P "N) ≡ ((P ") N)

3 Free and Bound Variables

In the expression X$_1$ ≡ ((λ ". (P" N)) O) ...

... the λ marks " the variable " as a parameter. In order to is variable " (the argument O) **tied** (*bound*)
However, the variables P , N and O are **released** (*free*).

Calculate the **set of free / bound variables** in a λ-expression:

PY " (B) = { B }
PY " (('$_1$' $_2$)) = PY " ('$_1$) ∪ PY' ' (' $_2$)
PY " ((λB.'$_1$)) = PY " ('$_1$) \ { B } λB binds B □

^ _`ab (B) = ∅
^ _`ab (( '$_1$' $_2$ )) = ^ _`ab ('$_1$) ∪ ^ _`ab (' $_2$)
^ _`ab (( λB.'$_1$ )) = ^ _`ab ('$_1$) ∪ { B } λB binds B □

Free and bound variables (examples)

Example: Free variables in $X_1 \equiv ((\lambda ". (P" N)) O)$:

$$
\begin{aligned}
& PY " (((\lambda ". ((P") N)) O)) \\
&= PY " ((\lambda ". ((P") N))) \cup PY " (O) \\
&= (PY " (((P ") N)) \setminus \{ " \}) \cup \{ O \} \\
&= ((PY " ((P ")) \cup PY " (N)) \setminus \{ x \}) \cup \{ O \} \\
&= (((PY " (P) \cup PY " (x)) \cup \{ N \}) \setminus \{ " \}) \cup \{ O \} \\
&= (((\{ P \} \cup \{ " \}) \cup \{ N \}) \setminus \{ " \}) \cup \{ O \} \\
&= (\{ P , " , N \} \setminus \{ " \}) \cup \{ O \} \\
&= \{ P , N , O \} .
\end{aligned}
$$

Tie / freedom must for *every occurrence* of a

Variables are decided separately. **Example:**

$X_2 \equiv (\overline{"} (\lambda ".")) \; PY " (X_2) = \{ "^{\llcorner} \} \wedge \_`ab (X_2) = \{ " \}$

KK

free bound

4 ¦ Evaluation in the λ-calculus: β-reduction

The application of a function (= λ abstraction) to a

Argument is described by **β-reduction** $\rightsquigarrow_\beta$:

The application $((\lambda B.'_1) \; '_2)$ is carried out in the

1. a copy of the hull $'_1$ is made and
2. all **free occurrences of $ in the copy of the trunk**

   $\%_1$ **be** replaced by $'_2$.

$((\lambda ".P\_\_) \wedge XY)$ constant function

$\rightsquigarrow_\beta P\_\_$.

((λ ". ((λN. (* N")) X)) ^) ---------------Currying--------
⇝β ((λN. (* N ^)) X)
⇝β (* X ^) .

Evaluation in the λ-calculus: β-reduction

More examples of **β-reduction:**

bound free

H

((λ ". ((* ((λ". (+ "X)) ^)) . ")) 7) replace VE, F, "

k in the tru

hull

free

H

⇝β ((* ((λ ". (+" X)) ^)) 7) replace VE, F, "

in the trunk through ^

hull

⇝β ((* (+ ^ X)) 7) .

NB: Without a rule like apply_prim the reduction ends from the point of view of the λ-calculus here.

Evaluation in the λ-calculus: β-reduction and variable capture

Even more examples of **β-reduction:**

Type ⓘ

((λP. (P 7)) (λ ". (+" X))) Programming with HOF
⇝β ((λ ". (+" X)) 7)
⇝β (+ 7 X) .

ignore 2nd arg, return 1st arg

l———m————n

((((λ ". (λN.")) (λO.N)) ↯) X)

└---- i—j——k

ignore Arg, deliver N

would have to t
Deliver N ...

captured

⇝β (((λN. (ΛO.N)) ↯) X)
⇝β ((λO. ↯) X)
⇝β ↯ .

D: EF: <9, z: I +) E, ☐:
free N "wanders" under λN
and is bound with it

Evaluation in the λ-calculus: How *exactly* does β-reduction work ?

'{"→X} : " *In* " *, replace free occurrences of* # *with* $ " . In order to

then ((λ ". ') X) ⇝β ' {" →X} :

" { " →X } = X

B { "→X } = B

('₁' ₂) { "→X } = ('₁ { " →X } ' ₂ { "→X })

(λ ". '₁) { " →X } = (λ ".' ₁)

(λB.'₁) { "→X } = { (λB.'₁ { "→X }) B not free in X ※ [→5]
(λB '.' ₁ { B→B ' }) { "→X } otherwise [→

※ If B is not free in X , B cannot be captured
become. Name B ' is new (we need a pool of names).

Evaluation in the λ-calculus: β-reduction in the example

Three β-reductions evaluate ((((λ ". (ΛN.")) (ΛO.N)) ↯) X) : [1]

((((λ ". (λN.")) (λO.N)) ↯) X) expected result: N

⇝β (((λN. ") { " →(λO.N)} ↯) X) N free in (λO.N)
→6 (((λN '. " { N→N' }) { " →(λO.N)} ↯) X) N ': new nam
→2 (((λN '. ") { " →(λO.N)} ↯) X) N' not free in (λO.N)
→5 (((λN '. " { " →(λO.N)}) ↯) X)
→1 (((λN '. (ΛO.N)) ↯) X)

⇝β ((λO.N) { N'→↯ } X) O not free in ↯
→5 ((λO.N { N'→↯ }) X)
→2 ((λO.N) X)

⇝β N { O→X }
→2 N.

[1] ____ marks the expression to be reduced (*% & 'ucible & (pression , redex*).

5 ┆ Evaluation in the λ-calculus: normal form

Evaluation of an expression ' in the λ-calculus: Iterate β-Reduction until 'has been converted to its **normal form** :

'is in Ü * E;: 9V * E; ⇔ ∄ " ∊> : '⇝β' '

'cannot be reduced any further

Examples:

" is in NF

(λ" "." ") is in NF
((λ " . " ") O) is not in NF
((λ ". (" ")) (λ". (""))) is not in NF
(and doesn't have any)

Is it OK to speak of *the* (unambiguous) normal form?

Evaluation in the λ-calculus: The normal form is unambiguous

Example: Reduce the two *redexes* in the order ①②
and ②①. Does the end result of the reduction differ?

①

((λO. (OX)) ((λ ". (λN.")) ^)) ②å$_\beta$ ((λO. (OX)) (λN. ^))

②

①h$_\beta$ h$_\beta$

(((λ ". (λN.")) ^) X) å$_\beta$ ((λN. ^) X)

h$_\beta$

^ in NF

Coincidence? No! □ Church-Rosser theorem.

Evaluation in the λ-calculus: Church-Rosser theorem

Church-Rosser theorem: If the expression ' is (in several)
Steps (⇝$_\beta$ *) to '$_1$ and '$_2$, then there are

an expression '' in which $'_1$ and $'_2$ can be reduced:

β * $'_1$
è ê$_β$ *
' ''
ê è$_β$ *
β * $'_2$

Church-Rosser “Diamond” ◇

[without pr

Consequence: Should $'_1$ *and* $'_2$ be in normal form, then applies $'_1$ = ' = $'_2$ ($'_i$ ⇝$_β$ * '' performs zero-β reductions out). So $'_1$ = '' = $'_2$ is ***the*** normal form of ' .

6 ┊ Reduction strategy (*Redex* selection) *Applicative Order*

Applicative order evaluates λ-expression ' by repeating β-Reduction off. $[\![']\!]^k$ with maximum i is the next *redex* :

- $[\![B]\!]^k \overset{def}{=} B$ [: *
- $[\![(\lambda B.'_1)]\!]^k \overset{def}{=} (\lambda B.[\!['_1]\!]^k)$ [
- $[\![('_1 '_2)]\!]^k \overset{def}{=}$ Let $P \equiv [\!['_1]\!]^{k+1}$, then :

$$\begin{cases} [\![' \{ B \overset{*}{\to} [\!['_2]\!]^{k+2} \}^{k+1}]\!]^k & P = (\lambda B. ') \text{ [: * Û} \\ (P\ [\!['_2]\!]^{k+1}) & \text{otherwise [: *:]} \end{cases}$$

- *Applicative order* first evaluates the **innermost** *Redex* from (see rule: * û, ∗). This makes argument $'_2$ evaluated *before* the function application takes place.

Applicative order reduction *strategy* can fail

Certain expressions ' finds *applicative order* the Normal form of the expression *not* . **Example:** reduction of ((λ ".N) Ω) :2

1. Reduction via *Applicative Order* :

$$[\![((\Lambda\ ".N)\ \Omega)\ ]\!]^0 = \cdots = [\![N\ \{\ " \to [\![\Omega]\!]^2\ \}\ ^1]\!]^0$$

↻ £ endless reduction, since $[\![\Omega]\!]^k = [\![\Omega]\!]^k$

2. Reduce function application first (*normal order*):

$$\underline{((\lambda\ ".N)\ \Omega)} \rightsquigarrow_\beta N.$$

2 Ω ≝ ((λ ". (" ")) (Λ". (""))) .

7 ¦ Programming in the λ-calculus: *Booleans*

Q: Literals and operations on them are not in the λ-calculus to disposal. Can you ever program with it? !

A: Yes! Define λ-expressions which, **when interacting, the show expected algebraic properties** . **Example:**

§ • ¶> ≝ (λ ". (ΛN.")) ☐ ® © -> ™ ´> ≝ (λ ".")
© ¨ ™ ´> ≝ (λ ". (ΛN.N)) ☐ ¨ ≠ Æ ≝ (λX. (Λ ^. ((^ X) ^
≠ Ø§ ≝ ☐ exercise ☐ O• ≝ ☐ exercise

These λ-expressions behave like the Booleans:

⟦ (((® © -> ™ ´> § • ¶>) N'∞) a_) ⟧⁰ = N'∞
⟦ (((® © -> ™ ´> © ¨ ™ ´>) N'∞) a_) ⟧⁰ = a_
⟦ ((¨ ≠ Æ © ¨ ™ ´>) ") ⟧⁰ = © ¨ ™ ´>
⟦ ((¨ ≠ Æ § • ¶>) ") ⟧⁰ = "

Page 21

8 ¦ Programming in the λ-calculus: pairs, selectors and lists

Representation of **pairs** ⟨", N⟩ in the λ-calculus:

≥¨® • ≝ (λ ". (ΛN. (Λ∞. ((∞") N)))) ⟨", N⟩
´ ≠ Æ ≝ (λ ¥. (¥ (λ ". (ΛN.N)))) (´ ≠ Æ ⟨", N⟩) ⇝
© ´§ ≝ (λ ¥. (¥ (λ ". (ΛN.")))) (© ´§ ⟨", N⟩) ⇝'

i—j—k i—j—k
§ • ¶> ¥

Representation of **lists** (make-pair " ∞) and empty :

⟨© ¨ ™ ´>, ⟨"," ∞⟩⟩
l——m——

μ¨∂> -≥¨® • ≝ (λ ". (λ" ∞. ((≥¨® • © ¨ ™ ´>) ((≥¨® • ") " ∞))))
© ® • ´§ ≝ (λ "∞. (© ´§ (´ ≠ Æ" ∞)))
•> ´§ ≝ (λ "∞. (´ ≠ Æ (´ ≠ Æ" ∞)))
> μ≥§∑ ? ≝ © ´§
> μ≥§∑ ≝ ® (> μ≥§∑ ?> μ≥§∑) ⇝ (® § • ¶>) ⇝ § • ¶

Page 22

9 ¦ Programming in the λ-calculus: *Church Numerals*

Does a representation of the **natural numbers** { 0 , 1 , 2 ,

…} In the λ-calculus, which allows arithmetic operations?

Definition: ã is the ***Church Numeral*** for a ∊ ℕ:

$$\tilde{a} \overset{def}{=} (\lambda P. (\lambda ''. (P^{n}'')))$$
$$= (\lambda P. (\lambda ''. (\underbrace{P \cdots (P (P'')) \cdots}_{\text{a-fold application of P}})))$$

Examples:

$$\tilde{0} \equiv (\lambda P. (\Lambda ''.''))$$
$$\tilde{1} \equiv (\lambda P. (\Lambda ''. (P'')))$$
$$\tilde{2} \equiv (\lambda P. (\Lambda ''. (P (P''))))$$
$$\vdots$$

Page 23

" is the **fixed point** of function P if P (") = " applies.

Examples (real functions):

P (") =" 2	two fixed points: "= 0," = 1
P (") =" + 1 no fixed point	
P (") ="	an infinite number of fixed points

☐ In the λ-calculus **every function 'has a fixed point:** (∑ ©) .

Let $\sum \overset{def}{=} (\lambda P. ((\Lambda ''. (P ('' ''))) (\lambda''. (P (''''))))))$. Then:

$$(© (\sum ©)) = (\sum ©)$$ (∑ ©) is the fixed point of ©

Page 24

Recursion in the λ-calculus: The (combiner

(∑ ©)

= ((λP. ((λ ". (P (" "))) (λ". (P (""))))) ©)

(YF) = (F (YF))

(YF) =

(Y

⇝β ((λ ". (© (" "))) (λ". (© (""))))

⇝β (© ((λ ". (© (" "))) (λ". (© ("")))))

WITH S

⇜β (© ((λP. ((Λ ". (P (" "))) (λ". (P (""))))) ©))

= (© (∑ ©))

To ⇜β: The following applies: ' = ((λP.' {© →P}) ©) (β-abstraction).

∑ is Haskell B. Curry's (**combiner** that we use can to express **recursion in the λ-calculus** .

Page 25

Recursion in the λ-calculus: example factorial function)!

λ-term (© w a) calculates the **Faculty**)! **recursive:**

© ¨Ω ≝ (λa. (((® © -> ™ ´> (®´æ> • Ø? a)) recursion
ø̃)
¿
((µ¶ ™ § a) (© ¨Ω (≥ •> Æ a)))))

⇜β ((λP. (Λa. (((® © -> ™ ´> (®´æ> • Ø? A))
ø̃)
((µ¶ ™ § a) (P (≥ •> Æ a)))))) © ¨Ω

i ——— j
≡ ©

It follows that © ¨Ω = (© © ¨Ω) ⇒ © ¨Ω is the fixed point of © .

☐ Define the factorial function as © ¨Ω ≝ (∑ ©) .

☐ Recursive P. ☐ © ≡ β-abstraction from P. ☐ P ≝ (∑ ©).

Page 26

Recursion in the λ-calculus: A reduction strategy for (

⚠ *Applicative order* leads to endless reduction for (∑ ©) :

$$((\underline{\Sigma\ ©}\ )\ \widetilde{a_0}\ ) \rightsquigarrow ((\ ©\ (\underline{\Sigma\ ©}\ ))\ \widetilde{a_0}\ ) \rightsquigarrow ((\ ©\ (\ ©\ (\ \Sigma\ ©\ \underline{)))\ \widetilde{a_0}\ )} \rightsquigarrow \cdots \infty$$

Required: a reduction strategy, the **application of functions** reduced **before function** arguments. Approximately ***normal order*** : 3

$$\underline{((\ \Sigma\ ©\ )}\ \widetilde{a_0}\ ) \rightsquigarrow ((\ ©\ \underline{(\ \Sigma\ ©\ ))\ \widetilde{a_0}\ )} \rightsquigarrow \text{© decides} \begin{cases} \underline{'_1\ [}((\ \Sigma\ ©\ )\ \widetilde{a_1}\ )] \rightsquigarrow \cdots \\ \underline{'_2\ [\ \widetilde{a_0}\ ]}\ .\ \text{Recursion terr} \end{cases}$$

3 $'_1$ ['] denotes an expression $'_1$ in which ' occurs as a partial expression.

Page 27

11 ┆ Reduction strategy *Normal Order* 4

☐ ⟦B⟧^{k} ≝ B [G

☐ $[\![(\lambda B.'_1)]\!]^k \overset{def}{=} (\lambda B. [\!['_1]\!]^k)$ [

☐ $[\![('_1 \, '_2)]\!]^k \overset{def}{=}$ Let $P \equiv ⦇'_1⦈^{k+1}$, then :

$$\overset{*}{\begin{cases} [\![' \{ B \to '_2 \}^{k+1}]\!]^k & P = (\lambda B.\, ') \; [\, G * û \,] \\ ([\![P]\!]^{k+1} \, [\!['_2]\!]^{k+1}) \text{ otherwise} & [\, G *\text{:} \end{cases}}$$

Internally, rule G * û uses the *Call by Name* Reduction $⦇⦈^k$:

☐ $⦇B⦈^k \overset{def}{=} B$ [<

☐ $⦇ (\lambda B.'_1) ⦈^k \overset{def}{=} (\lambda B.'_1)$ [

☐ $⦇ ('_1 \, '_2) ⦈^k \overset{def}{=}$ Let $P \equiv ⦇'_1⦈^{k+1}$, then :

$$\overset{*}{\begin{cases} ⦇ ' \{ B \to '_2 \}^{k+1} ⦈^k & P = (\lambda B.\, ') \; [\, <G \, û \,] \\ (P \, '_2) & \text{otherwise} \quad [\, <G\text{:} \end{cases}}$$

4 Available as function no in the file definitions-13.rkt .