λ

Lambda Calculus - Part I

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Organization information

Next weeks

- May 14, 2024
  - a seminar by a colleague working with functional programming languages

- May 16, 2024 – ML Challenge
  - Teams of 3 students or for students who cannot physically attend teams of one student
  - Program in ML
  - I will send you a form for registering the group next week
  - An evaluation committee will evaluate your work

Next weeks

- We have to skip a couple of lessons – we have to find another day:

- Monday (e.g., May 20) 11:30 – 13:30
- Friday (e.g., May 24) 15:30 – 17:30
- Wednesday May 22 8:30 – 10:30

Intermediate feedback form

- Please, fill in the form at

https://forms.gle/i7mH13wRDv61etyN6

- It will remain open one week more

Final Exam

- In two parts
  - Multiple choice (or few open questions) exam on the topics of the theory part of the course (50%). Passing this part is required to take the second part
  - Programming problem(s) in ML. (50%).


TEST

Simulation

- One of the last classes, we will have the exam simulation

Today

- Recap
- Lambda calculus
  - Introductory concepts
  - Beta-reductions
  - Alpha-equivalence

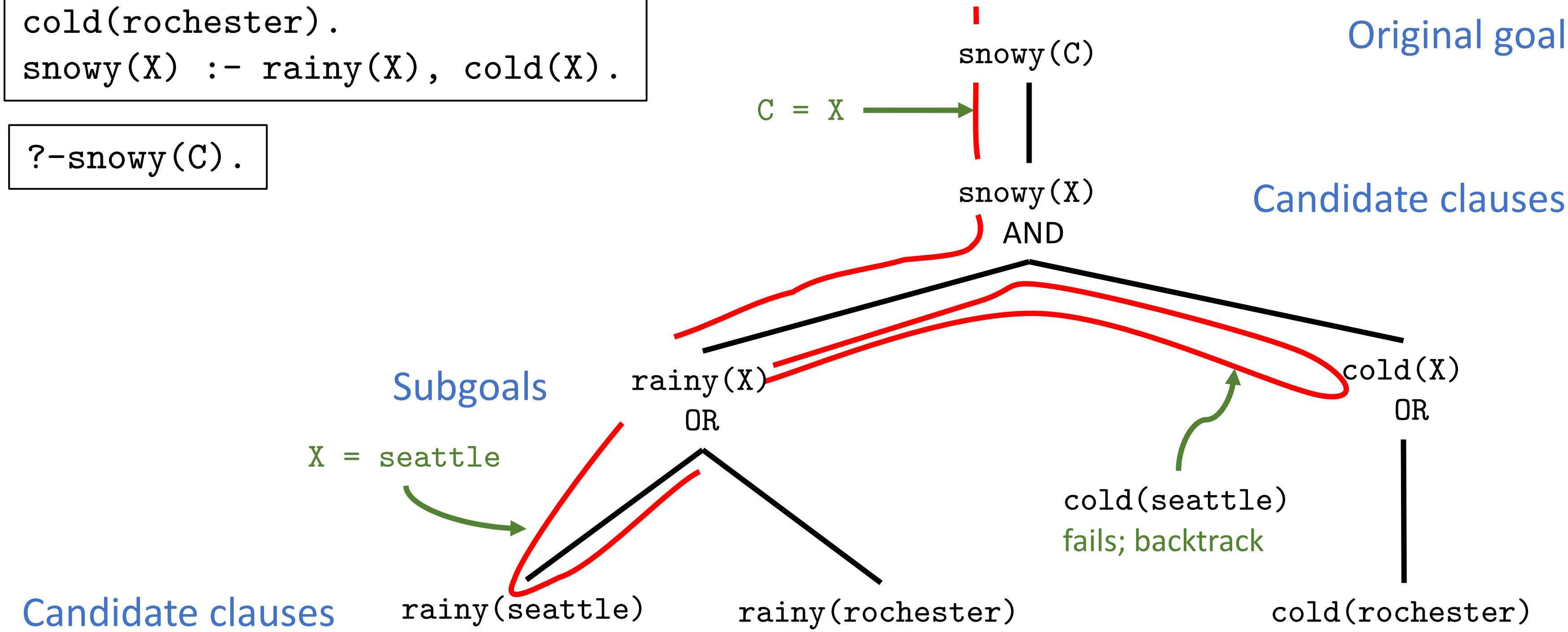
Recap

The Prolog interpreter explores the tree depth first, from left to right .

Search order

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

```
?-snowy(C).
```

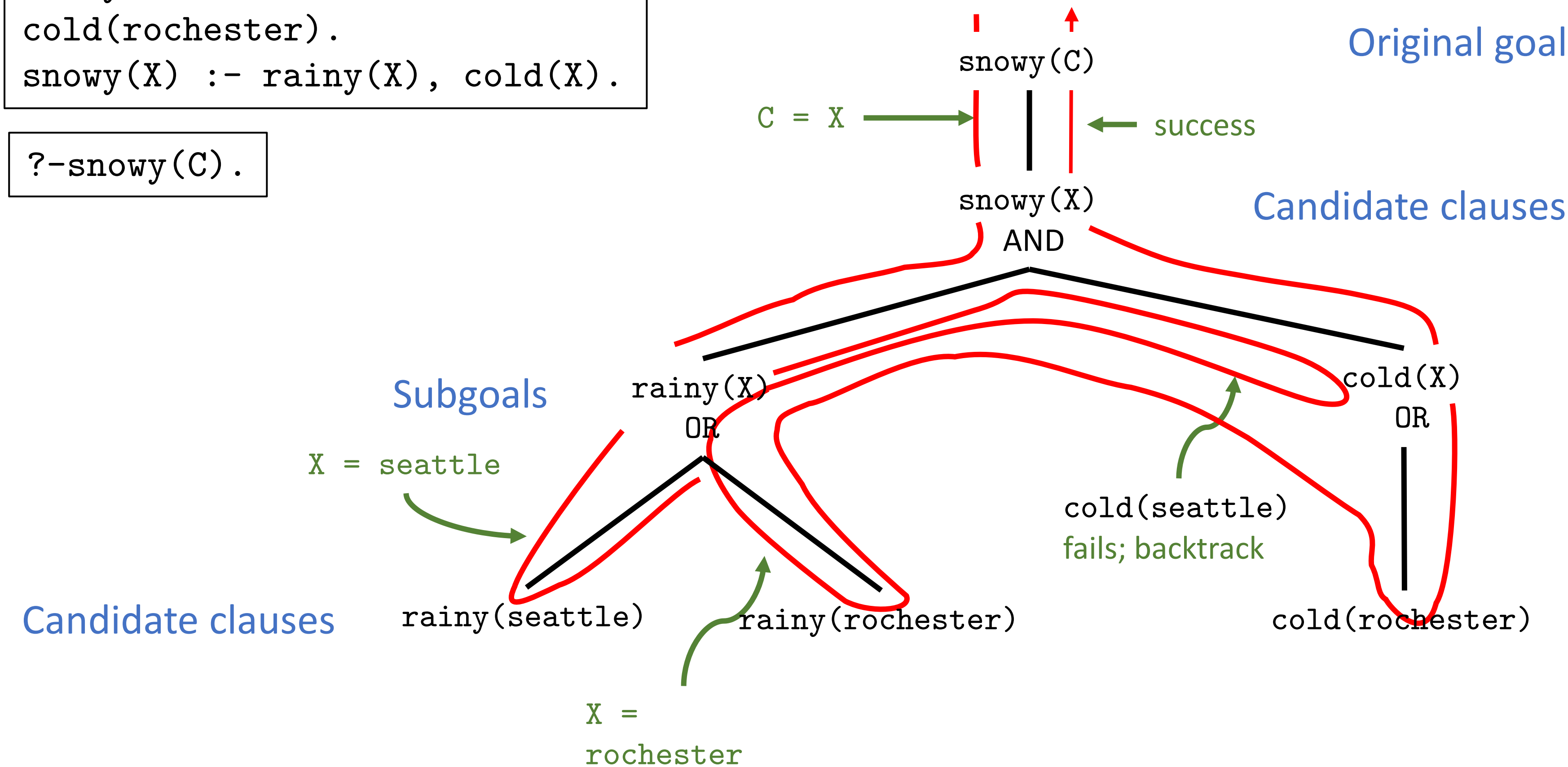


Search order

If at any point a subgoal fails (cannot be satisfied), the interpreter returns to the previous subgoal and attempts to satisfy it in a different way (i.e., try to unify it with the head of a different clause).

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

```
?-snowy(C).
```

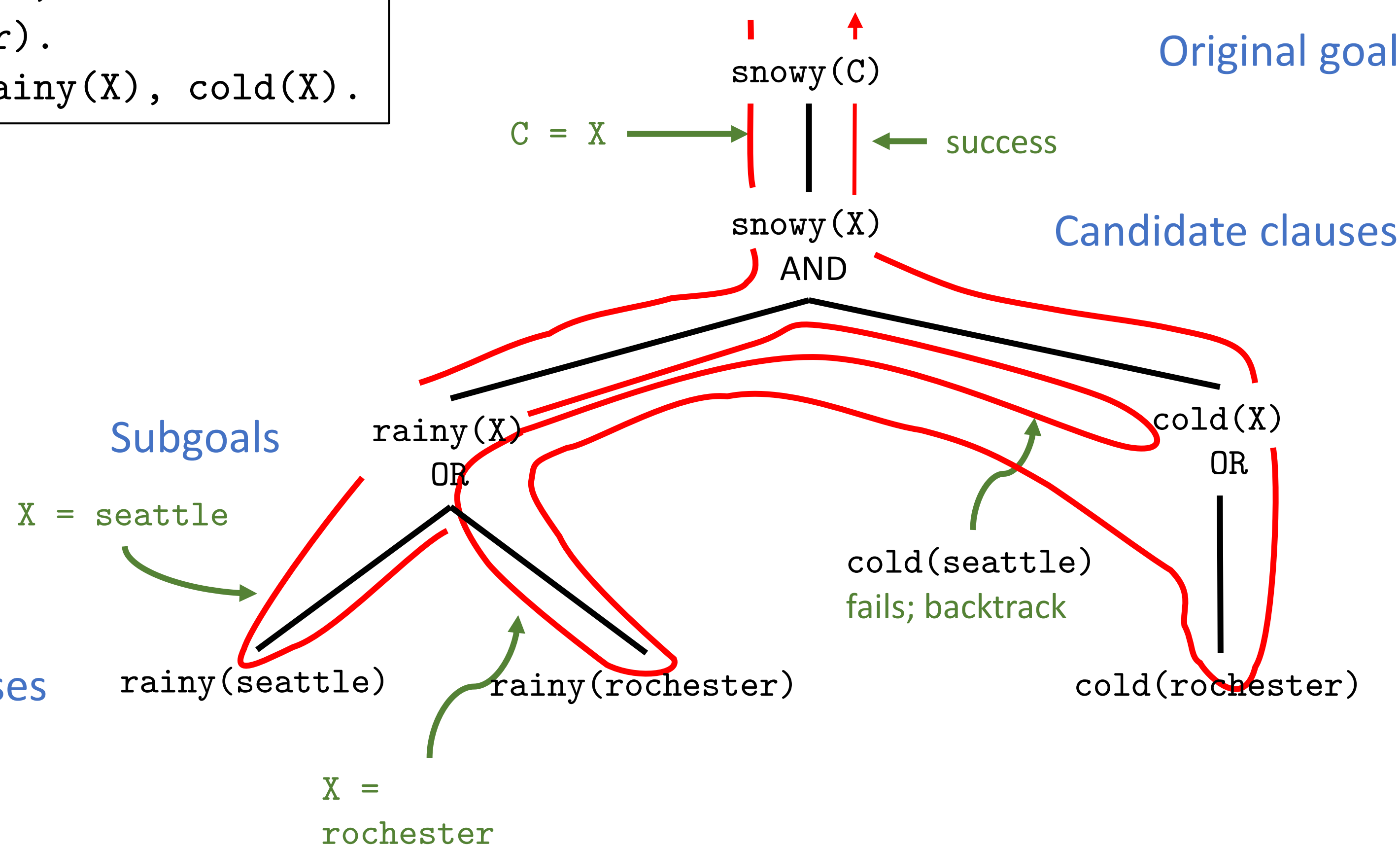


Search order

```
[trace]  ?- snowy(C).
   Call: (10) snowy(_18738) ? creep
   Call: (11) rainy(_18738) ? creep
   Exit: (11) rainy(seattle) ? creep
   Call: (11) cold(seattle) ? creep
   Fail: (11) cold(seattle) ? creep
   Redo: (11) rainy(_18738) ? creep
   Exit: (11) rainy(rochester) ? creep
   Call: (11) cold(rochester) ? creep
   Exit: (11) cold(rochester) ? creep
   Exit: (10) snowy(rochester) ? creep
C = rochester.
```

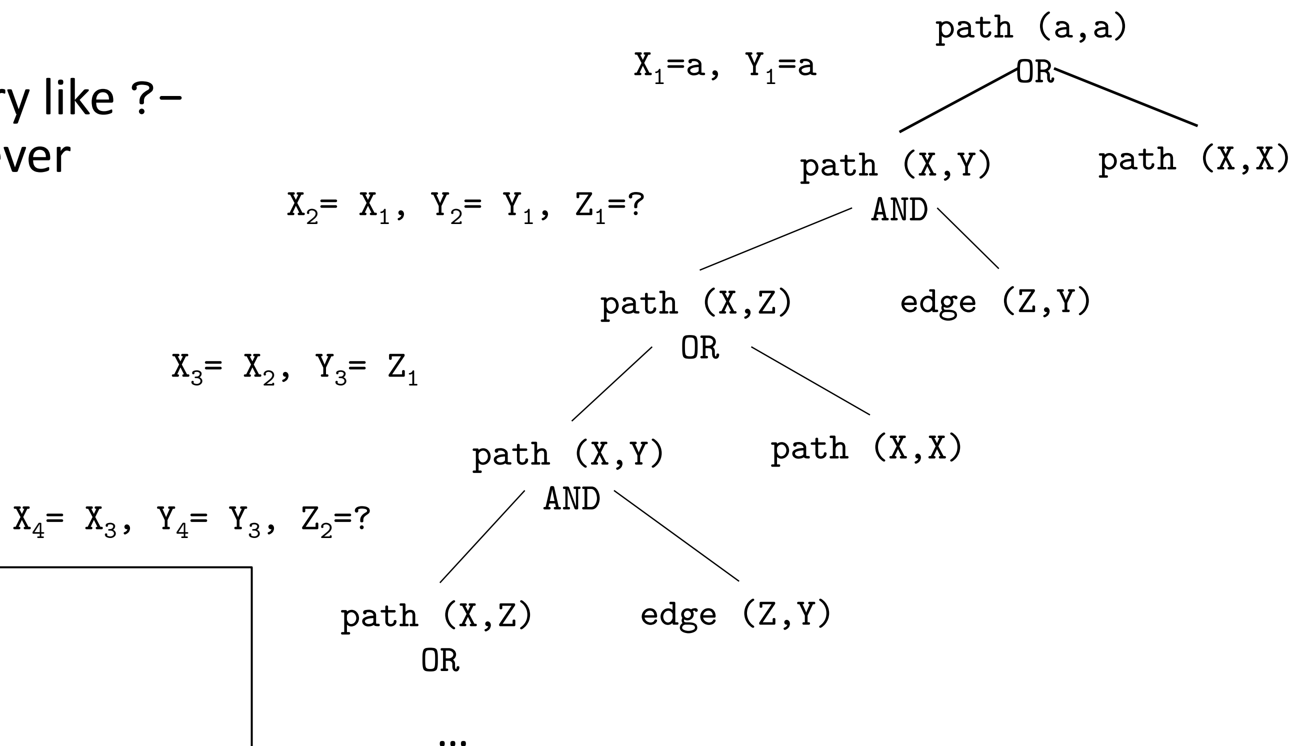
```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

```
?-snowy(C).
```



Predefined order of exploration: non-termination

Even a simple query like `?- path(a,a)` will never terminate




```
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(b, e).
edge(d, f).
path(X, Y) :- path(X, Z),
    edge(Z,Y).
path(X, X).
```

The ordering of clauses and of terms in Prolog is significant, with consequences for efficiency, termination, and choice among alternatives

Cut

- It allows the programmer to eliminate some of the possible alternatives produced during evaluation, so as to improve efficiency
- It is a zero-argument predicate written as an exclamation point: !
- As a subgoal it always succeeds, but it cannot be backtracked

The cut

- In general, if we have *n* clauses to define the predicate *p*

```
p(S1) :- A1.
...
p(Sk) :- B, !, C.
...
p(Sn) :- An.
```

- When we try to apply *k*th clause in the, we have the following cases:
  1. If the evaluation of B fails, then we proceed by trying the *k* +1st clause.
  2. If the evaluation of B succeed, then ! is evaluated. It succeeds and the evaluation proceeds with C. In case of backtracking, however, all the alternative ways of computing B are eliminated, as all the alternatives provided by the clauses from the *k*th to the *n*th to compute `p(t)`.

Why using cuts?

- Save time and space, or eliminate redundancy
  - Prune useless branches in the search tree
    - If sure these branches will not lead to solutions
  - These are green cuts
- Guide the search to a different solution
  - Change the meaning of the program
    - Intentionally returning only subsets of possible solutions
  - These are red cuts

Other predicates

- The negation (`not`) predicate applied to `G`, tries to satisfy it.
  - If `G` fails, then `not(G)` succeeds
  - If `G` succeeds, then `not(G)` fails
  - If `G` does not terminate, then `not(G)` negation as failure
- The `if-then-else` predicate: `B -> C1;C2`.
  - It is actually defined as

```
if-then-else(B,C1,C2) :- B, !, C1.
if-then-else(B,C1,C2) :- C2.
```

Other predicates

- The `call` predicate takes a term as argument and attempts to satisfy it as a goal
  - E.g., `call(makemove).`
- The `fail` predicate always fails
  - E.g., `fail.`
- The `repeat` predicate can succeed an arbitrary number of times
  - E.g., `repeat, write(a).`
- The `assert` predicate allows to add facts and rules to the database
  - E.g., `assert(rainy(syracuse)).`
- The `retract` predicate allows us to remove facts and rules from the database
  - E.g., `retract(rainy(rochester)).`

`fail` and `cut`

- In some cases, we may have a generator that produces an unbounded sequence of values
- The following generates all of the natural numbers.

```
natural(1).
natural(N) :- natural(M), N is M+1.
```

```
?- natural(X).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 ;
X = 6 ;
X = 7 ;
X = 8 ;
X = 9 ;
X = 10 ;
X = 11
```

- We can use this generator with a “cut-fail” combination to iterate over the first *n* numbers:

```
my_loop(N) :- natural(I), I =< N, write(I), nl, I = N,
!, fail.
```

- As long as I is lower than *N*, the equality predicate will fail, and backtracking will pursue another alternative
- If *I = N* succeeds, the `cut` is executed, committing the current (final) choice of *I* and terminating the loop.

```
?- my_loop(3).
1
2
3
false.
```

Characteristics of logic programming

- We only need to define the logic specifications
- Pros of “computation as deduction”
  - ability to use a program in more than one way (both input and output)
  - Possibility of obtaining several solutions
  - Possibility of looking at a program as a logical formula
- Cons
  - Backtracking can be inefficient
  - Absence of types and modules
  - Not very well developed programming environments

λ

Lambda-calculus

What is the lambda-calculus?

- A very simple, but Turing complete, programming language
  - created before concept of *programming language* existed!
  - helped to define what *Turing complete* means!

The lambda calculus

- Originally, the lambda calculus was developed as a logic by Alonzo Church in 1932
  - Church says: "There may, indeed, be other applications of the system than its use as a logic."

Early history of the lambda calculus

- Meanwhile, in England …
  - young Alan Turing invents the Turing machine
- Turing heads to Princeton, studies under Church
  - prove lambda calculus, Turing machine, general recursion are equivalent – they define the class of computable functions
  - Church–Turing thesis: these capture all that can be computed

The λ-calculus

- Purpose: formal mathematical basis for functional programming
- Why lambda? Evolution of notation for a bound variable:
  - Whitehead and Russell, *Principia Mathematica*, 1910

    $2\hat{x} + 3$ – corresponds to $f(x) = 2x + 3$
  - Church's early handwritten papers

    $\hat{x}: 2x + 3$ – makes scope of variable explicit
  - Typesetter #1

    $\wedge x: 2x + 3$ – couldn't typeset the circumflex!
  - Typesetter #2

    $\lambda x.2x + 3$ – picked a prettier symbol

Barendregt, *The Impact of the Lambda Calculus in Logic and Computer Science*, 1997

Impact of the lambda calculus

- Turing machine: theoretical foundation for imperative languages
  - Fortran, Pascal, C, C++, C#, Java, Python, Ruby, JavaScript, . . .
- Lambda calculus: theoretical foundation for functional languages
  - Lisp, ML, Haskell, OCaml, Scheme/Racket, Clojure, F#, Coq, . . .

The λ-calculus

1. Introduces variables ranging over values – e.g., $x + 1$
2. Define functions by (lambda-)abstracting over variables –e.g., $\lambda x. x + 1$
3. Apply functions to values – e.g., $(\lambda x. x + 1)2$

For instance we can write a function (computing the square of a variable) without naming it

$$(\lambda x. x^2)$$

and we can apply the function to another expression

$$(\lambda x. x^2)7 = 49$$

Formally

- When dealing with λ-calculus, given a countable set of variables V, we have

$$e ::= x \mid \lambda x. e \mid e\ e$$

that is, an expression e can be

- x: a variable $\in V$
- $\lambda x. e$: a function taking as input a parameter x and evaluating the expression e (abstraction)
- $e\ e$: the application of two expressions

Lambda calculus and ML syntax

λ-Calculus syntax

- $\lambda x. e$
- x: bound variable
- e: expression

ML syntax

- `fn x => e`
- `x`: formal parameter
- `e`: expression usually using `x`

Two operations

- Abstraction of a term with respect to a variable x: $\lambda x. e$, that is the function that when applied to a value v produces e in which v replaces x
- Application of a function to an argument: $e_1 e_2$, that is the application of the function e_1 to the argument e_2

Terminology

binder

$$\lambda x.e$$

The scope of x is the term e – body of the function

x is bound

$$\lambda x.xy$$

y is free

$$(\lambda x.\lambda y.zx(yx))x$$

Free and bound variables

- The set of free variables of an expression is defined by:
  - $F_v(x) = \{x\}$
  - $F_v(\lambda x. e) = F_v(e)\backslash\{x\}$
  - $F_v(e_1 e_2)= F_v(e_1) \cup F_v(e_2)$

  e.g., $F_v(\lambda x. y(\lambda y. xyu)) = \{y, u\}$
- The set of bound variables of an expression is defined by
  - $B_v(x) = \emptyset$
  - $B_v(\lambda x. e) = \{x\} \cup B_v(e)$
  - $B_v(e_1 e_2)= B_v(e_1) \cup B_v(e_2)$

  e.g., $B_v(\lambda x. y(\lambda y. xyu)) = \{x, y\}$

The abstraction (λ) operator removes a variable from the list of free variables and adds it to the bound ones

Conventions

- Associativity of application is on the left (as in ML)

  $y\ z\ x$ corresponds to $(y\ z)x$
- Parenthesis can be used for readability – though not strictly needed
  - $(((f_1 f_2) f_3) f_4)$ is more clear than $f_1 f_2 f_3 f_4$
- The body of a lambda extends as far as possible to the right, that is

  $\lambda x. x\ \lambda z. x\ z\ x$ corresponds to $\lambda x. \left(x\ \lambda z. (x\ z\ x)\right)$ and not to $\cancel{(\lambda x. x)\ (\lambda z. (x\ z\ x))}$
- Consecutive abstractions can be uncurried:

$$\lambda xyz. e = \lambda x. \lambda y. \lambda z. e$$

Curried functions

- Functions in ML have only one argument
- Functions with two arguments can be implemented as
  - A function with a tuple as argument
  - Curried form
    - Unary function takes argument x
    - The result is a function $f(x)$ that takes argument y

- Curried function: divides its arguments such that they can be partially supplied producing intermediate functions that accept the remaining arguments

Example

```
> fun exponent1 (x,0) = 1.0
    | exponent1 (x,y) = x * exponent1 (x,y-1);
val exponent1 = fn: real * int -> real

> fun exponent2 x 0 = 1.0
    | exponent2 x y = x * exponent2 x (y-1);
val exponent2 = fn: real -> int -> real

> exponent1 (3.0,4);
val it = 81.0: real

> exponent2 3.0 4 ;
val it = 81.0: real
```

-> associates to right:
`real -> (int -> real)`
`exponent2` is a function taking a `real` and returning a function from `int` to `real`

Partial instantiation

- Curried functions are useful because they allow us to create partially instantiated or specialized functions where some (but not all) arguments are supplied.

```
> val g = exponent2 3.0;
val g = fn: int -> real

> g 4;
val it = 81.0: real

> g (4);
val it = 81.0: real
```

We are partially instantiating exponent2 (with name g) – g is the power function with base 3.0

Question 1

$(\lambda x. y)\ z$ and $\lambda x. y\ z$ are equivalent

A. true.
B. false.

Answer question 1

$(\lambda x. y)\ z$ and $\lambda x. y\ z$ are equivalent

A. true
B. false

Question 2

$\lambda x. x\ a\ b$ is equivalent to?

A. $(\lambda x. x)\ (a\ b)$
B. $((\lambda x. x)a)\ b)$
C. $\lambda x. (x\ (a\ b))$
D. $(\lambda x. \big((x\ a)b\big))$

Answer question 2

$\lambda x. x\ a\ b$ is equivalent to?

A. $(\lambda x. x)\ (a\ b)$
B. $((\lambda x. x)a)\ b)$
C. $\lambda x. (x\ (a\ b))$
D. $(\lambda x. \big((x\ a)b\big))$

β

Beta-reduction

Lambda expression evaluation

The intuition

- Consider this lambda expression:

$$(\lambda x. x + 1)4$$

It means that we apply the lambda abstraction to the argument 4, as if we apply the increment function to the argument 4.

- How do we do it?

  The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

- This means: $(\lambda x. x + 1)4 \underset{\beta}{\rightarrow} 4 + 1$

β-reduction examples

- $(\lambda x. x + x)5 \rightarrow 5 + 5 \longrightarrow 10$
- $(\lambda x. 3)5 \rightarrow 3$

Parameters

- formal
- formal occurrence
- actual

It looks like we instantiate the formal parameter (i.e., the occurrences of the bound variable) with the actual parameter (the expression to which we are applying the function)

β-reduction examples

- $\underbrace{(\lambda x.(\lambda y.y - x))\ 4\ 5}$ $\rightarrow (\lambda y.y - 4)5 \rightarrow 5 - 4 \rightarrow 1$

  We can see this as currying –
  we peel off the argument 4 and then 5

- $(\lambda f.f\ 3)(\lambda x.x + 1) \rightarrow (\lambda x.x + 1)3 \rightarrow 4$

Parameters
- formal
- formal occurrence
- actual

β-reduction examples

- $(\lambda x.\, x)z \;\to z$
- $(\lambda x.\, y)z \;\to y$
- $(\lambda x.\, x\ y)z \;\to z\ y$
- $(\lambda x.\, x\ y)(\lambda z.\, z) \to (\lambda z.\, z)y \to y$
- $(\lambda x.\, \lambda y.\, x\ y)z \;\to \lambda y.\, z\ y$

a curried function of two arguments: it applies its first argument to its second

Parameters

- formal
- formal occurrence
- actual

β-reduction examples

- $(\lambda x.\lambda y.x\ y)(\lambda z.zz)x \to (\lambda y.(\lambda z.zz)y)x$
  $\to (\lambda z.zz)x \to xx$
- $\big(\lambda x.x\ (\lambda y.y)\big)(u\ r) \ \to (u\ r)(\lambda y.y)$
- $\big(\lambda x.(\lambda w.x\ w)\big)(y\ z) \to \lambda w.(y\ z)w$

Parameters
- formal
- formal occurrence
- actual

β-reduction examples

- $(\lambda x.\lambda z.x\ z)y$

$\rightarrow (\lambda x.(\lambda z.(x\ z)))y$ since λ extends to right

$\rightarrow (\lambda x.(\lambda z.(x\ z)))y$ apply $(\lambda x.e_1)e_2 \rightarrow e_1[e_2/x]$
where $e_1 = \left(\lambda z.(x\ z)\right), e_2 = y$

$\rightarrow \lambda z.(y\ z)$

Question 3

$\lambda x. y\ z$ can be beta-reduced to?

$A.\ y$
$B.\ y\ z$
$C.\ z$
D. cannot be reduced

Answer question 3

$\lambda x. y\ z$ can be beta-reduced to?

$A.\ y$
$B.\ y\ z$
$C.\ z$
D. cannot be reduced

Summary

- Lambda calculus

Next time

- More on lambda calculus