INF121:
Functional Algorithmic and Programming
Lecture 6: Polymorphism, Higher-order, and Currying

Academic Year 2011 - 2012

\[ f(x) \]
In the previous episodes of INF 121

- Basic Types: Booleans, Integers, Floats, Chars, String
- *if* ... *then* ... *else* ... conditional structure
- identifiers (local and global)
- defining and using functions
- Advanced types: synonym, enumerated, product, sum
- Pattern matching on simple and advanced expressions
- Recursion
  - recursive functions and their termination
  - recursive types and how to use them (in (recursive) functions + in pattern matching)
- List (recursive type) with Cons/Nil and OCaml’s pre-defined notations
Outline

Polymorphism

Higher-Order

Currying
Outline

Polymorphism

Higher-Order

Currying
Motivating polymorphism on examples
Limitations of Functions

About the *identity* function:

- **Identity on** `int`:
  ```plaintext
  let id (x:int):int = x
  val id:int → int = <fun>
  ```

- **Identity on** `float`:
  ```plaintext
  let id (x:float):float = x
  val id:float → float = <fun>
  ```

- **Identity on** `char`
  ```plaintext
  let id (x:char):char = x
  val id:char → char = <fun>
  ```
Motivating polymorphism on examples

Limitations of Functions

About the *identity* function:

- **Identity on int:**
  ```
  let id (x:int):int = x
  val id:int → int = <fun>
  ```

- **Identity on float:**
  ```
  let id (x:float):float = x
  val id:float → float = <fun>
  ```

- **Identity on char**
  ```
  let id (x:char):char = x
  val id:char → char = <fun>
  ```

Disadvantages:

- **1 function per type** needing the identity function
- Unique/Different names needed if these functions should “live” together
Motivating polymorphism on examples
Limitations of Functions on list

Compute the length of a list:

- of int:

  ```ml
  let rec length_int (l:int list):int= 
  match l with 
  | [] → 0 
  | _::l → 1+ length_int l
  ```

- on char:

  ```ml
  let rec length_char (l:char list):int= 
  match l with 
  | [] → 0 
  | _::l → 1+ length_char l
  ```

- ...

Remark: The body of these functions is not specific to char nor int → we need lists that are not bound to a type
Motivating polymorphism on examples

Limitations of Functions on list

Compute the length of a list:

- **of int:**

  ```ml
  let rec length_int (l:int list):int=
    match l with
    | [] → 0
    | _::l → 1+ length_int l
  ```

- **on char:**

  ```ml
  let rec length_char (l:char list):int=
    match l with
    | [] → 0
    | _::l → 1+ length_char l
  ```

- ... 

**Remark** The body of these functions is not specific to char nor int → we need lists that are *not bound to a type*
Motivating polymorphism on examples
Limitations of (current) lists

Several sorts of lists, à la Lisp:

- **type** listofint = Nil | Cons int * listofint
  and then Cons (2, Cons (9,Nil))

- **type** listofchar = Nil | Cons char * listofchar
  and then Cons ('t', Cons ('v',Nil))
Motivating polymorphism on examples

Limitations of (current) lists

Several sorts of lists, à la Lisp:

- **type** listofint = Nil | Cons int * listofint
  and then Cons (2, Cons (9,Nil))

- **type** listofchar = Nil | Cons char * listofchar
  and then Cons ('t', Cons ('v',Nil))

Several sorts of lists, even with OCaml shorter notations:

- list of int: [1;2] (=1::2::[ ]) of type int list
- list of char: ['e'; 'n'] (= b :: ' n :: [ ]) of type char list
- list of string: ["toto";"titi"] (="toto"::"titi"::[ ]) of type string list
Back on the examples - introducing polymorphism

Let’s come back on the (various) identity functions

What if we omit type?

```
let id x = x

val id : 'a -> 'a = <fun>
```
Back on the examples - introducing polymorphism

Let’s come back on the (various) identity functions

What if we omit type?
```ocaml
let id x = x
val id: 'a -> 'a = <fun>
```

→ type inference: OCaml computes the most general type
→ **polymorphic** identity: the id on any type (\(\alpha\) or ‘a)

“id is a polymorphic function” that can be applied to any value

We can specifically indicate that the function can take any type:

```
let id (x: 'a): 'a = x
```
equivalently
```
let id (x: 'b): 'b = x
```
equivalently
```
let id (x: 'toto): 'toto = x
```

... ...

→ the type returned by OCaml is ‘a -> ‘a
(equivalently \(\alpha \rightarrow \alpha\))

**DEMO: Polymorphic identity**
Polymorphic lists

We can define lists that are parameterized by some type (à la Lisp)

\[
\text{type } t\text{ llist } = \text{Nil} \mid \text{Cons of } t \ast t\text{ llist}
\]

\(t\) is a type parameter

OCaml pre-defined lists are already parameterized by some type:

- type of \([\ ]\) is \(\alpha\text{ list}\) (equivalently \(\alpha\text{ list}\))
- type of \(::\) is \(\alpha \rightarrow \alpha\text{ list} \rightarrow \alpha\text{ list}\)
  (equivalently \(\alpha \rightarrow \alpha\text{ list} \rightarrow \alpha\text{ list}\))

Remark  Still, the elements should have the same type
Polymorphic lists

We can define lists that are parameterized by some type (à la Lisp)

\[
\text{type } \tau \text{llist} = \text{Nil} | \text{Cons of } \tau \times \tau \text{llist}
\]

\(\tau\) is a type parameter

OCaml pre-defined lists are already parameterized by some type:

- type of [] is \(\alpha\)list (equivalently \(\alpha\)list)
- type of :: is \(\alpha \to \alpha\)list \(\to \alpha\)list (equivalently \(\alpha \to \alpha\)list \(\to \alpha\)list)

Remark Still, the elements should have the same type

Example

- Cons (2,Cons (3,Cons (4,Nil)))
- Cons ('r',Cons ('d',Cons ('w',Nil)))
- Cons ((fun x \to x), Cons ((fun x \to 3x+2), Nil))
Polymorphic functions on Polymorphic lists

Let’s practice

Example (Length of a list)
À la Lisp:

```ocaml
let rec length (l:'a llist):int =
  match l with
  | Nil → 0
  | Cons (_, l) → 1 + length l
```

OCaml pre-defined lists:

```ocaml
let rec length (l:'a list):int =
  match l with
  | [] → 0
  | _::l → 1 + length l
```
Polymorphic functions on Polymorphic lists
Let’s practice

Example (Length of a list)
À la Lisp:

```
let rec length (l: 'a llist): int =
    match l with
    | Nil → 0
    | Cons (_, l) → 1 + length l
```

OCaml pre-defined lists:

```
let rec length (l: 'a list): int =
    match l with
    | [] → 0
    | _::l → 1+ length l
```

Exercise: implement some functions on polymorphic lists

- **isEmpty**: returns **true** if the argument list is empty
- **append**: appends two lists together
- **reverse**: reverse the elements of a list
- **separate**: inserts a separator between two elements of a list

Note: be careful with the types
Outline

Polymorphism

Higher-Order

Currying
Consider two simple functions returning the maximum of two integers:

\[
\text{let max2\_v1 } (a:\text{int})(b:\text{int}):\text{int} = \begin{align*}
\quad & \text{if } a \geq b \text{ then } a \\
\quad & \text{else } b
\end{align*}
\]

\[
\text{let max2\_v2 } (a:\text{int})(b:\text{int}):\text{int} = \begin{align*}
\quad & \text{if } a \leq b \text{ then } b \\
\quad & \text{else } a
\end{align*}
\]
Consider two simple functions returning the maximum of two integers:

```plaintext
let max2_v1 (a:int)(b:int):int =
  if a >= b then a
  else b

let max2_v2 (a:int)(b:int):int =
  if a <= b then b
  else a
```

Several questions:

- How to test whether those functions are correct?
- How to test whether those functions return the same values for the same input values?
Consider two simple functions returning the maximum of two integers:

```plaintext
let max2_v1 (a:int)(b:int):int = if a >= b then a else b

let max2_v2 (a:int)(b:int):int = if a <= b then b else a
```

Several questions:

- How to test whether those functions are correct?
- How to test whether those functions return the same values for the same input values?

**DEMO**: Higher-order can provide elegant testing solutions
Introduction to higher-order

In OCaml and functional programming, functions is the basic tool:
- to “slice” a program into smaller pieces
- to produce results

Functions are first-class citizens: they are values (e.g., used in lists, . . .)

A function can also be a parameter or a result of a function

Example (Returning an affine function)
```ocaml
let affine a b = (fun x → a * x + b)
```

Several benefits
Definition (Higher-Order language)

A programming language is a higher-order language if it allows to pass functions as parameters and functions can return a function as a result.

Remark  The C programming language allows to pass functions as parameters but does not allow to return a function as a result.
Higher-order functions
Some vocabulary

Definition (Higher-Order language)
A programming language is a higher-order language if it allows to pass functions as parameters and functions can return a function as a result.

Remark: The C programming language allows to pass functions as parameters but does not allow to return a function as a result.

Definition (Higher-order function)
A function is said to be a higher-order function or a functional, if it does at least one of the two things:

▶ take at least one function as a parameter
▶ return a function as a result

Remark: Non higher-order functions are said to be first-order functions.
Higher-order functions
Benefits and What you should learn

Conciseness

Some form of expressiveness

At the end of the day, you should know:

- that higher-functions exist
- the associated "vocabulary"
- know how and when to use it

We will demonstrate and experiment those features through examples...
Example (Slope of a function in 0)
Let $f$ be a function defined in 0 (with real values):

$$\frac{f(h) - f(0)}{h} \quad \text{(with } h \text{ small)}$$

**DEMO: Slope in 0**

Example (Derivating a (derivable) function $f$)
We approximate $f'(x)$ (value of the derivative function in $x$) by:

$$\frac{f(x + h) - f(x)}{h} \quad \text{(with } h \text{ small)}$$

**DEMO: Derivative**
Reminders:

- A zero of a function $f$ is an $x$ s.t. $f(x) = 0$
- Theorem of *intermediate values*:
  Let $f$ be a continuous function, $a$ and $b$ two real numbers, if $f(a)$ and $f(b)$ are of opposite signs, then there is a zero in the interval $[a, b]$
- $\sqrt{a}$ is the positive zero of the function $x \mapsto x^2 - a$
- $\forall a \geq 0 : 0 \leq \sqrt{a} \leq \frac{1 + a}{2}$
A tour of some higher-order functions

Numerical functions

Reminders:

- **A zero** of a function $f$ is an $x$ s.t. $f(x) = 0$
- **Theorem of intermediate values:**
  Let $f$ be a continuous function, $a$ and $b$ two real numbers, if $f(a)$ and $f(b)$ are of opposite signs, then there is a zero in the interval $[a, b]$
- $\sqrt{a}$ is the positive zero of the function $x \mapsto x^2 - a$
- $\forall a \geq 0 : 0 \leq \sqrt{a} \leq \frac{1 + a}{2}$

Exercise: zero of a continuous function using dichotomy

- Define a function `sign` indicating whether a real is positive or not
- Deduce a function `zero` that returns the zero of a function, up to some given epsilon, given two reals s.t. there is a zero between those reals
- Deduce a function to approximate the square root of a float
A tour of some higher-order functions

Applying twice a function

Consider the two functions `double` and `square`:

- `let double (x:int):int = 2*x`
- `let square (x:int):int = x*x`
Consider the two functions `double` and `square`:

- `let double (x:int):int = 2*x`
- `let square (x:int):int = x*x`

How can we define `quad` and `power4` reusing the previous function?

- `let quad (x:int):int = double (double x)`
- `let square (x:int):int = square (square x)`
A tour of some higher-order functions
Applying twice a function

Consider the two functions `double` and `square`:

- `let double (x:int):int = 2*x`
- `let square (x:int):int = x*x`

How can we define `quad` and `power4` reusing the previous function?

- `let quad (x:int):int = double (double x)`
- `let square (x:int):int = square (square x)`

Can we generalize?... Yes, we can:

`let applyTwice (f:int → int) (x:int):int = f (f x)`

- `let quad (x:int):int = applyTwice double x`
- `let power4 (x:int):int = applyTwice square x`

or using anonymous functions:

- `let quad (x:int):int = applyTwice (fun (x:int) → 2* x) x`
- `let power4 (x:int):int = applyTwice (fun (x:int) → x * x) x`
A tour of some higher-order functions

Composing functions

Function composition:

\[ f : C \rightarrow D \]
\[ g : A \rightarrow B \]
\[ g \circ f : C \rightarrow B \quad \text{if } D \subseteq A \]

Let us simplify and take \( D = A \), hence \( g \circ f : C \xrightarrow{f} A \xrightarrow{g} B \)
A tour of some higher-order functions

Composing functions

Function composition:

\[ f : C \rightarrow D \]
\[ g : A \rightarrow B \]
\[ g \circ f : C \rightarrow B \quad \text{if } D \subseteq A \]

Let us simplify and take \( D = A \), hence \( g \circ f : C \xrightarrow{f} A \xrightarrow{g} B \)

Exercise: Defining function composition in OCaml

» Specify the function \( \text{compose} \) that composes two functions (beware of types)

» Implement the function \( \text{compose} \)

In OCaml:
if \( f \) is a function of type \( \text{t1} \rightarrow \text{t2} \) and \( g \) a function of type \( \text{t2} \rightarrow \text{t3} \) then

» \( \text{compose} \ g \ f \) will be of type \( \text{t1} \rightarrow \text{t3} \)

» \( \text{compose} \) will be of type \( (\text{t2} \rightarrow \text{t3}) \rightarrow (\text{t1} \rightarrow \text{t2}) \rightarrow (\text{t1} \rightarrow \text{t3}) \)

DEMO: Implementation of \( \text{compose} \)
Consider a series defined as follows:

\[
\begin{align*}
  u_0 &= a \\
  u_n &= f(u_{n-1}), \quad n \geq 1
\end{align*}
\]

The \(n\)-th term \(u_n\) is \(f(u_{n-1}) = f(f(u_{n-2})) = f(f(f(\ldots (u_0) \ldots)))\).
Consider a series defined as follows:

\[
\begin{align*}
    u_0 &= a \\
    u_n &= f(u_{n-1}), n \geq 1
\end{align*}
\]

The \(n\)-th term \(u_n\) is \(f(u_{n-1}) = f(f(u_{n-2})) = f(f(f(\ldots(u_0)\ldots)))\)

**Exercise: \(n\)-th term of a series**

Define a function \texttt{nthterm} that computes the \(n\)-th term of a series defined as above using a function \(f\) and some \(n\)

**Exercise: \(n\)-th iteration of a function**

Define a function \texttt{iterate} that computes the function which is the \(n\)-th composition of a function, given some \(n\)
A tour of some higher-order functions
Generalizing the sum of the $n$ first integers

Sum of $n$ first integers:

$$1 + 2 + \ldots + (n - 1) + n = (1 + 2 + \ldots + (n - 1)) + n$$

Implemented as:

```ocaml
let rec sum_integers (n:int) =
  if n=0 then 0 else sum_integers (n-1) + n
```

Sum of the integers through a function - generalization

▶ Define a function $\sigma$ that computes the sum of the images through some function for the first $n$ integers

▶ Give an alternative implementation of `sum_integers` and `sum_squares` using $\sigma$
A tour of some higher-order functions

Generalizing the sum of the $n$ first integers

Sum of $n$ first integers:

$$1 + 2 + \ldots + (n - 1) + n = (1 + 2 + \ldots + (n - 1)) + n$$

Implemented as:

```ocaml
let rec sum_integers (n:int) =
  if n=0 then 0 else sum_integers (n-1) + n
```

The sum of squares is similarly:

$$1^2 + 2^2 + \ldots + (n - 1)^2 + n^2 = (1^2 + 2^2 + \ldots + (n - 1)^2) + n^2$$

Implemented as:

```ocaml
let rec sum_squares (n:int) =
  if n=0 then 0 else sum_squares (n-1) + (n*n)
```
A tour of some higher-order functions

Generalizing the sum of the $n$ first integers

Sum of $n$ first integers:

$$1 + 2 + \ldots + (n-1) + n = (1 + 2 + \ldots + (n-1)) + n$$

Implemented as:

```ml
let rec sum_integers (n:int) =
    if n=0 then 0 else sum_integers (n-1) + n
```

The sum of squares is similarly:

$$1^2 + 2^2 + \ldots + (n-1)^2 + n^2 = (1^2 + 2^2 + \ldots + (n-1)^2) + n^2$$

Implemented as:

```ml
let rec sum_squares (n:int) =
    if n=0 then 0 else sum_squares (n-1) + (n*n)
```

Sum of the integers through a function - generalization

- Define a function $\sigma$ that computes the sum of the images through some function for the first $n$ integers
- Give an alternative implementation of $\text{sum\_integers}$ and $\text{sum\_squares}$ using $\sigma$
Another representation of the list $l = [a_1; a_2; ...; a_n]$:
A tour of some higher-order functions
Lists: applying a function on all elements on a list - function \texttt{map}

Given:

- a list of type `\texttt{a list}`
- a function of type `\texttt{a → b}`

\[
\begin{align*}
\text{Given:} & \quad \begin{cases} 
\text{a list of type } \texttt{a list} \\
\text{a function of type } \texttt{a → b}
\end{cases} \\
\text{map } f \ l & \quad \Rightarrow \\
\end{align*}
\]
A tour of some higher-order functions
Lists: applying a function on all elements on a list - function map

Given:
- a list of type \( 'a \) list
- a function of type \( 'a \to 'b \)

\[
\begin{align*}
\text{map } f & \text{ l} \\
\text{a}_1 & \to f \text{ a}_1 \\
\text{a}_2 & \to f \text{ a}_2 \\
\ldots & \to \ldots \\
\text{a}_n & \to f \text{ a}_n
\end{align*}
\]

Remark
- Application of \( f \) does not depend on the position of the element
- \( \text{map} \) returns a list
- \( \text{map} \) can change the type of the list

Typing
If \( l \) is of type \( t_1 \) list and \( f \) is of type \( t_1 \to t_2 \) then \( \text{map } f \text{ l} \) is of type \( t_2 \) list
A tour of some higher-order functions
Lists: applying a function on all elements on a list - function map

Exercise: function map

Define a function map such that:

- given a list and a function \( f \) on the elements of that list,
- returns the list where \( f \) has been applied to all elements of that list
A tour of some higher-order functions

Lists: applying a function on all elements on a list - function map

Example (Vectorize)

- Specification:
  - Profile: `vectorize: Seq(Elt) → Vec(Seq(Elt))`, where `Vec` is the set of lists of one element
  - Semantics:
    \[ \text{vectorize} [e_1;...;e_n] = [[e_1]; ... ;[e_n]] \]

- Implementation:
  \[
  \text{let vectorize } = \text{my_map (fun } e \rightarrow [e])
  \]

Example (Concatenate to each)

- Specification:
  - Profile: `Seq(Elt) * Seq(Seq(Elt)) → Seq(Vec(Elt))`
  - Semantics:
    \[ \text{concatenate_to_each} (l, [v_1; ...; v_n] = [ l@v_1; ... ; l@v_n ] \]

- Implementation:
  \[
  \text{let concatenate_to_each } = \text{fun } (l,seqv) \rightarrow \text{my_map (fun } x \rightarrow l@x) \text{ seqv}
  \]
A tour of some higher-order functions
Lists: applying a function on all elements on a list - function map

Exercise: using the function map for converting lists

Define the following functions:
- toSquare: raises all elements of a list of int to their square
- toAscii: returns the ASCII code of a list of char
- toUpperCase: returns a list of char where all elements have been put to uppercase

Exercise: Powerset

Define the function powerset that computes the set of subsets of a set represented by a list
A tour of some higher-order functions
Lists: iterating a function on all elements on a list - function fold_right - some intuition first

Example (Sum of the elements of a list)

```ml
let rec sum l =
    match l with
    [] → 0
    | elt::remainder → elt + (sum remainder)
```

Example (Product of the elements of a list)

```ml
let rec product l =
    match l with
    [] → 1
    | elt::remainder → elt * (product remainder)
```

Example (Paste the string of a list)

```ml
let rec concatenate l =
    match l with
    [] → " "
    | elt::remainder → elt ^ (concatenate remainder)
```

Remark  Notice that the only elements that change are:
- the “base case”, i.e., what the function should return on the empty list
- “how we combine the current element with the result of the recursive call
A tour of some higher-order functions
Lists: iterating a function on all elements on a list - function \texttt{fold_right}

If we place the operator in prefix position, we have:

- \texttt{sum} \([a_1;a_2;...;a_n] = + a_1 (+ a_2 (... (+ a_n 0)...))
- \texttt{product} \([a_1;a_2;...;a_n] = * a_1 (* a_2 (... (* a_n 0)...))
- \texttt{concatenate} \([a_1;a_2;...;a_n] = ^ a_1 (^ a_2 (... (^ a_n 0)...))
A tour of some higher-order functions

Lists: iterating a function on all elements on a list - function fold_right

If we place the operator in prefix position, we have:

- sum \([a_1;a_2;...;an] = + a_1 (+ a_2 (... (+ an 0)...))\)
- product \([a_1;a_2;...;an] = \ast a_1 (\ast a_2 (... (\ast an 0)...))\)
- concatenate \([a_1;a_2;...;an] = ^{a_1} (^{a_2} (... (^{an} 0)...))\)

More generally, given:

- \(f\) of type \(’a \rightarrow ’b \rightarrow ’b\),
- \(l\) of type \(’a\) list, and
- some initial value \(b\) of type \(’b\)

```
fold_right f l b
```

\rightarrow \text{result is of type ’a}
Exercise: writing \texttt{fold\_right}

Given

- \( f \) of type \( 'a \rightarrow 'b \rightarrow 'b \), and
- \( l = [a_1;...;a_n] \) of type \( 'a \text{ list} \),

define a function \texttt{fold\_right} s.t.

\[
\text{fold\_right } f \,[a_1;...;a_n] \,b = f \,(a_1 \,(... \,f \,(a_n \,b)))
\]

Exercise: using \texttt{fold\_right}

- Re-write the previously defined functions, \texttt{sum}, \texttt{product}, \texttt{concatenate} using \texttt{fold\_right}

- Define a function that determines whether the number of elements of a list is a multiple of 3 without using the function returning the length of a list
A tour of some higher-order functions
A small case-study with fold_right

Exercise: tasting testing

The purpose is to write a test suite function
We have seen examples of test cases
A test suite is a series of test cases s.t.:
▶ each test case is applied in order
▶ for a test suite to succeed, all its test cases must succeed

Questions:
▶ Define a function test_suite that checks whether two functions \( f \) and \( g \) returns the same values on a list of inputs values. Each element of the list is an input to the two functions.
▶ Here are two simple functions:
  ▶ let plus1 = fun x -> x + 1
  ▶ let plus1dummy = fun x -> if (x mod 2 = 0) then x - 2 + 3 else 2*x

Find 2 lists of inputs, so that the application of the function test_suite
  1. finds the bug
  2. does not find the bug
A tour of some higher-order functions
Lists: iterating a function on all elements on a list - function fold_left

More generally, given:
- f of type 'a → 'b→ 'a,
- l of type 'b list, and
- some initial value a of type 'a:

\[
\begin{align*}
\text{fold}_\text{left} & \quad f \quad a \quad l \\
\text{result is of type } & \quad 'a
\end{align*}
\]
A tour of some higher-order functions
Lists: some function parameterized by a predicate

A predicate is a function that returns a Boolean

Recall the function that removes not even integers from a list of integers:

```ocaml
let rec remove_odd (l:int list) =
  match l with
  | [] → []
  | elt::remainder →
    if elt mod 2 = 0
      then elt::(remove_odd remainder)
    else (remove_odd remainder)
```

A tour of some higher-order functions
Lists: some function parameterized by a predicate

Exercise: Filtering according to a predicate
Define a function \texttt{filter} that filters the elements of a list according to some given predicate \( p \)

Exercise: Checking a predicate on the elements of a list

- Define a function \texttt{forall} that checks whether \textit{all} the elements of a list satisfy a given predicate \( p \)
- Define a function \texttt{exists} that checks whether \textit{at least one} element of a list satisfy a given predicate \( p \)
A tour of some higher-order functions
Some more exercises

Exercise: back to testing

- Redefine the function `test_suite` using the function `forall`

Exercise: Map with fold

- Redefine `map` using `fold_left`
- Redefine `map` using `fold_right`

Exercise: minimum and maximum with one line of code

Define the functions `minimum` and `maximum` of a list using `fold_left` and `fold_right`. The function can be written with one line of code
Outline

Polymorphism

Higher-Order

Currying
About Currying

A function with $n$ parameter $x_1, \ldots, x_n$ is actually a function that takes $x_1$ as a parameter and returns a function that takes $x_2, \ldots, x_n$ as parameters.
About Currying

A function with \( n \) parameter \( x_1,\ldots,x_n \) is actually a function that takes \( x_1 \) as a parameter and returns a function that takes \( x_2,\ldots,x_n \) as parameters.

The application

\[
  f \ x_1 \ x_2 \ \ldots \ \ x_n
\]

is actually a series of applications

\[
  f \ (\ldots \ (f \ x_1) \ x_2) \ \ldots \ ) \ x_n
\]

**Definition: Partial application**

Applying a function with \( n \) parameters with (strictly) less than \( n \) parameters.

The result of a partial application remains a function.

**Typing:**

If

- \( f \) is of type \( t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \), and
- \( x_i \) is of type \( t_i \) for \( i \in [1,j] \subseteq [1,n] \)

Then \( f \ x_1 \ x_2 \ \ldots \ x_j \) is of type \( t_{(j+1)} \rightarrow \ldots \rightarrow t_n \rightarrow t \).
Example (Apply twice)

Back to the function applyTwice:

```ml
let applyTwice (f:int → int)(x:int):int = f (f x)
```

Applying `applyTwice` with only one argument:

```ml
applyTwice (fun x → x +4)
```

is equal to the function

```ml
fun x → x + 8
```
Currying has some advantages

Suppose we want a function taking \( a \in A \) and \( b \in B \) and returning \( c \in C \)

Without currying: \( f : tA \times tB \rightarrow tC \)
- \( f \) takes 1 argument: a pair
- \( f(a,b) \) is of type \( tC \)

With currying: \( f : tA \rightarrow tB \rightarrow tC \)
- \( f \) takes 2 arguments
- \( f(a)(b) \) is of type \( tC \)

\[ f(a) \] is of type \( tB \rightarrow tC \)

DEMO: 2 definitions of integer addition & the predefined (+) in OCaml

Lessons learned

- Currying allows some flexibility
- Allows to specialize functions

Remark  When applying curried functions, it can be harder to detect that we have forgot a parameter
Conclusion / Summary

Polymorphism

- general types
- "type parameterization"

Higher-Order

- "taking a function as a parameter or returning a function"
- improve conciseness, expressiveness, quality,…

Currying

- partial application of a function
- function specialization
- define your function so it can be curried