Functional Programming: Spicing it Up

Luca Abeni

luca.abeni@santannapisa.it

Function and... Spices???

- How are functions (in particular, pure functions) related to spices???
  - In no way... Here, “Curry” is not a spice
  - Let’s see...
- What are we going to talk about?
  - Functions. Functions having multiple formal parameters
  - λ calculus only considers functions with a single argument $\Rightarrow$ some functional programming languages allow to define single-argument functions
  - How to implement a function like $f(a, b) = a^2 + b^2$?

An Example

- Multivariable functions: let's try to understand them
- Function "`sum2`" implementing $f(x, y) = x^2 + y^2$
- From a matemathical point of view, $f : \mathcal{N}^2 \rightarrow \mathcal{N}$
- It can be implemented as a function with a couple of integers as its single argument:

```
int sum2(std::pair<int, int> v)
{
  return v.first * v.first + v.second * v.second;
}
```

- Can we do this without using structured data types as formal parameters?
  - No "`std::pair<>`" or similar, only scalar types!

From the Mathematical Point of View

- Functions like $f : \mathcal{N}^2 \rightarrow \mathcal{N}$ requires structured data types (a tuple, in this case) for the parameter
- Alternative: we need two arguments, but we can have only one... Let's return a function that receives the second argument!
  - Instead of having (x, y) as an argument and returning $x^2 + y^2$, let's have x as an argument and return a function that receives y as an argument and returns $x^2 + y^2$!
  - The function is now $\mathcal{N} \rightarrow (\mathcal{N} \rightarrow \mathcal{N})$
  - Simple, no?

Currying

- Currying: generic technique used to transform a multivariable function in a "chain of functions" with a single argument
  - Comes from Haskell Curry (mathematician), not from Masala Curry (spice)...
- Currying transforms $f(x, y) : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$ into $f_c(x) = C(f) : \mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{C})$ (often written $\mathcal{A} \rightarrow \mathcal{B} \rightarrow \mathcal{C}$)...
  - ...So that $(f_c(x))(y) = f(x, y)$
  - Note: "$f_c(x)$" is a function of y... We can have $g = f_c(x)$, with $g(y) = f(x, y)$!
- This also works with more than 2 arguments

Mathematically Speaking...

- Since Haskell Curry was a mathematician...
  - ...Let's try to formalize the currying mechanism from a mathematical point of view!
- Set F of functions $f : \mathcal{D} \rightarrow \mathcal{C}$: $\mathcal{F} = \mathcal{C}^{\mathcal{D}}$ (set of subsets of $\mathcal{D} \times \mathcal{C}$)
- For two-variables functions, $\mathcal{D} = \mathcal{A} \times \mathcal{B}$: $f(x, y) : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$... $\mathcal{F} = \mathcal{C}^{\mathcal{A} \times \mathcal{B}}$
  - Instead of $f : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$ we can use $f : \mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{C})$
- Set $\mathcal{F}_c$ of functions from $\mathcal{A}$ to functions from $\mathcal{B}$ to $\mathcal{C}$: $\mathcal{F}_c = (\mathcal{C}^{\mathcal{B}})^{\mathcal{A}}$
- Currying can be seen as a mapping from $\mathcal{F}$ a $\mathcal{F}_c$ (which ensures that the final result is preserved)

Mapping Functions to Curried Functions

- Currying as a mapping / mathematical function
  - From the set $\mathcal{F}$ of functions $f(x, y) : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$
  - To the set $\mathcal{F}_c$ of functions $f_c(x) : \mathcal{A} \to (\mathcal{B} \to \mathcal{C})$

$$curry : \mathcal{C}^{\mathcal{A} \times \mathcal{B}} \to (\mathcal{C}^{\mathcal{B}})^{\mathcal{A}}$$

- Fundamental importance: we can consider only functions with a single scalar argument!
  - Ok, the return type is not scalar... :)

Practical Currying

- Some programming languages (example: ML) allow to define only functions with a single argument...
  - ...The currying mechanism shows that this is not a limitation!
  - And functions with multiple arguments can be encoded using currying
- We will see that this also happens with “λ”
- Simple example in Standard ML
  - `(fn x => fn y => x * x + y * y) a b` = `((fn x => fn y => x * x + y * y) a) b`
  - First, the “`fn x`” thing is applied to “`a`”, then the resulting function is applied to “`b`”!
- Can we do this in C++, too?

Currying in C++

```
int sum2(std::pair<int, int> v)
{
  return v.first * v.first + v.second * v.second;
}
```

becomes

```
std::function<int(int)> sum2_c(int a)
{
  return [a](int b) {
    return a * a + b * b;
  };
}
```

which can be invoked as "`f_c(a)(b)`"

Currying: not in C!

```
int (* sum2_c(int a))(int)
{
  int s(int b) {
    return a + b;
  };

  return s;
}
```

- OK, nested functions are a non-standard GNU extension...
- But, can you see other issues?
  - Hint: `sum2_c` here returns a function pointer, not a closure...

Exercizes

- Using C++ lambdas, write the curried form of:
  - The factorial function, with tail recursion
  - The GCD computation function
  - The function solving the problem of the Towers of Hanoi
- Look at the “derivative” and “compute derivative” examples again, and think again about differences and similiarities between the two functions