Functional Languages

Unipd 2023-2024

Contents

Intro

What makes a language functional? The fact that "*functions are first class citizen of the language*". This means that functions can be passed as arguments to other functions, returned as values from other functions, and assigned to variables or data structures.

Some basics

- **Statements** are single line of code of type `statement`
- **Expressions** are entities that can be evaluated and have a type

Let's describe (not completely) the `C` grammar in BNF notation:

```
expr ::= constant
      |  (expr)
      |  x
      |  expr++
      |  ++expr
      |  expr--
      |  --expr
      |  expr = expr
      |  expr + expr
      |  expr += expr
      |  expr - expr
      |  expr -= expr
      |  expr * expr
      |  expr *= expr
      |  expr / expr
      |  expr /= expr
      |  expr % expr
      |  expr %= expr
      |  expr << expr
      |  expr <<= expr
      |  expr >> expr
      |  expr >>= expr
      |  expr & expr
      |  expr &= expr
      |  expr | expr
      |  expr |= expr
      |  expr ^ expr
      |  expr ^= expr
      |  ~ expr
      |  - expr
      |  ! expr
      |  expr && expr
      |  expr || expr
      |  expr == expr
      |  expr < expr
      |  expr <= expr
      |  expr > expr
      |  expr >= expr
      |  & expr
      |  * expr
      |  expr ? expr : expr
      |  f(expr1, ..., exprN)
      |  expr[expr]
      |  expr.x
```

```
constant ::= 0 | 1 | 2 | 3 | ...
          |  'a' | 'b' | ...
          |  1.0 | 2.45 | ...
          |  "string..."

block ::= statement ;
       |  statement ; block

cases ::= case constant: block
       |   case constant: block cases

statement ::= return expr
           |  if (expr) { block }
           |  if (expr) { block } else { block }
           |  for (statement ; expr ; expr ) { block }
           |  while (expr) { block }
           |  do { block } while (expr)
           |  switch (expr) { cases }
           |  break
           |  continue
           |  goto x
```

NOTE: this notation is called *Backus-Naur Form.*

It is used to describe context-free grammars and syntax of languages.

Reduction

Reduction is the evaluation of an expression. With this process every part of the expression is "reduced" to a smaller form until we get to a *ground value*. Let's take $1 + 2 * 4$ as example:

$$\overbrace{\overbrace{\underbracket{\underbracket{1+2}_{3}*4}_{12}}^{\text{int}}}^{\text{int}}$$

On the lower brackets we can see the result of the evaluation of the arithmetical operations. From the upper brackets instead we can see that the reduction preserves the types of the expression.

Declaration, assegnation, binding

In imperative languages there are some core concepts like **declaration** and **assignment** or **initialiation**. So for example in `C`

```c
int x;              // declaration

x = 7;              // assignment

int answer = 42;    // declaration with initialiation
```

In the context of functional languages, following the syntax of mathematics, we use only the last option, declaration with initialization, and we call it *binding*.

Binding answer to 42 in F# will be `let answer = 42`

Syntax vs Semantics... and types

Let's take for example this code `int x = &42`, a declaration with inizialitation in `C`.

This respects the grammar of the language because it is **syntactically correct**, in fact this can be described as `Type ID = expr`. We can also expand that expression in `expr = &expr`, which is the address extraction of a variable, where the internal `expr` is the constant expressions `42`. But speaking about **semantics** this is **wrong**!

To understand why this is incorrect we have to analyze the types of the expression. The address extraction operation transform types like that $\&\tau \rightsquigarrow \tau *$, so in our example it transforms `int` to an `int*`.

Characteristics of functional languages

Functional languages:

- have only expressions
- have variable definition
- have function definition
  - Note: in the C grammar described before is missing function definition

They have to offer the minimum to be Turing-complete, so a way of looping is needed. This structure is missing in this type of languages, but covered by *recursion*.

Syntactical vs Lexical

Lexicon: collection of words. In programming language is the set of keywords and values that can be used.

So, for example, -7 is different from `-expr` where the expression is 7.

In `expr + expr` there are two expression and the operator + which is a keyword in the lexicon of the language.

Functional vs Imperative

We have to consider that "*to assing*" means to modify. This is why functional languages lacks of assignment operation, meaning anything that can modify data directly.

```
int x = 3;  // declaration with initialiation -> a BINDING in FL

x = 4;      // assignment -> MISSING in FL
```

Polymorphism

When the type of an expression is not important we can use some form of **polymorphism**. For example it could be **subtyping**, such as the possibility of interchange sub and base classes in OOP. Another way to implement polymorphism is by **parametric polymorphism**, with techniques like Java Generics and C++ Templates (this is the way F# follows).

Function applications

The line in the `C` grammar that allow us to do function calls is $f(e_1, ..., e_n)$. In functional languages we use to call this *application* of the function and usually it is called just by writing the name of the function followed by the arguments.

The "functional grammar" for the application is `expr ::= expr expr`.

Take for example the function `(fun x -> x) 7`. This is the identity function, so from something it returns the exactly same thing. In this case types are not explicit, so the compiler create some "generic" *anonymous types*.

`(fun x -> x) 7`	F# function
$f : \grave{}a \rightarrow \grave{}a$	Anonymous type generated by the compiler
$f : \mathrm{int} \rightarrow \mathrm{int}$	Types by inferred the compiler using the type of the argument

Note: in application the left part must be an arrow (a function). We can say that lambdas create and applications remove arrows.

After the application, the type is inferred by the compiler. It understands that we are passing a integer, so the function becomes "*monomorphic at time of application*" over the type `int`.

Binding of parameters

```
let succ x = x + 1  // Syntactic sugar for this: let succ = fun x -> x + 1

let seven = f 6     // 6 is bounded to x then the function is computed
```

That 6 in the function applications replaces every occurency of `x` in the scope of the parameter.

Errors accepted by the grammar

If we look at the grammar for function application application (but not only in that) we can see that it is possible to create something that is syntactically correct but that will produce an error. For this we need to use the power of **types** and **type check** what we create with our grammar.

Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Let's see an example with integer addition. The uncurried form of this operation is

```
// uncurry_add : int * int -> int
// Syntactic sugar for uncurry_add = fun (x, y) -> x + y
let uncurry_add (x, y) = x + y

let result = uncurry_add (7, 8)
```

It accepts only pair of integers as arguments and after the application it returns the result.

In its curried form instead we have

```
// curry_add : int -> (int -> int)
// Syntactic sugar for curry_add = fun (x, y) -> x + y
let curry_add (x, y) = x + y

// partial_application : int -> int
let partial_application = curry_add 7

// final_application : int
let final_application = z 1

// total_application : int -> int -> int
let total_application = curry_add 7 1
```

Currying is in fact using more arrows in a function definition. With this form we can pass single integers as arguments and make "partial" applications of the function. It is automatically applied associativity on the left.

Function transformer

We can define functions that transform from uncurried version to the curried one and viceversa.

```
// curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
let curry f x y = f (x, y)

// uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
let uncurry f (x, y) = f x y
```

`curry` converts an uncurried function to a curried function. `uncurry` converts a curried function to a function on pairs. Let's see an example with the addition:

```
>>> curry uncurry_add 40 2
>>> 42

>>> uncurry curry_add (2, 40)
>>> 42
```