

The Foundations of Functional Programming

Luca Abeni

`luca.abeni@santannapisa.it`

Programming Paradigms

- Programs can be developed using **many** different paradigms
 - Imperative: computation as state modification
 - Functional: computation as reduction (???)
 - ...
- Imperative paradigm
 - Mutable variables: **environment** associating names to variables, **store** associating variables to values
 - Assignments are the core of programs
 - Modify the store ($f: \text{variable} \rightarrow \text{value}$)
 - Each variable “contains” an R-value
 - Directly maps to Von Neumann machines

Functional Programming Paradigm

- Functional Programming → no state / mutable variables
 - No mutable variables \Rightarrow no assignments!
 - Environment without store
 - Programs composed by expressions and functions (no commands)
 - Computation as **reduction / substitution of expressions**
 - Instead of state mutation...
- Reduction??? WTH is this???
 - Replacing the invocation of a function with the returned value...

Functional Programming Technique 1: Recursion

- No mutable state \rightarrow no iteration (loop)!
 - Iteration is based on repeating something while a predicate is true
 - Predicate: boolean function of the state...
Immutable state \Rightarrow the predicate is always true or always false \Rightarrow infinite loop, or no iteration!
- Use **Recursion** instead of iteration!
 - Mathematical model: λ -calculus!

Mathematical Functions

- Function: relation between domain and codomain, associating at most an element of the codomain to each element of the domain
 - $f : \mathcal{X} \rightarrow \mathcal{Y}$
 - $f \subset \mathcal{X} \times \mathcal{Y} : (x, y_1) \in f \wedge (x, y_2) \in f \Rightarrow y_1 = y_2$
 - $(x, y) \in f \rightarrow y = f(x)$
- $f(x)$ is... Ambiguous?
 - $f(x) = x^2$: definition of $f()$
 - $f(3)$: application of $f()$ to 3
 - The same syntax ($f(x)$) is used for **definition** and **application** of a function?

Programming with Functions

- In math, the meaning of “ $f(x)$ ” depends on the context...
 - Example: “ $f(x) = x^2$ ” vs “ $f(3)$ ”
- ...A programming language needs a more univoque syntax!
 - We need a different syntax for application and definition
- Some examples:
 - C/C++: “ $\{ . . . \}$ ” after the function’s prototype is used for definitions
 - In ML, fn is used to define a function
 - ...

Function Definitions vs Expressions

- Special syntax to *define* functions
 - In C, “`double f(double x) {return x * x; }`” defines $f(x) = x^2$
 - But... This is not an expression!!!
- Strange idea: use expressions to define functions...
Something like “`f1 = {return x * x; }`”???
- Not possible in C... Functions are not expressible or storable values...
- ...Maybe, we can store/express *function pointers* but not functions!
- In C++, “`auto f1=[](double x){return x*x;};`”
- Notice: these are **real functions**, not function pointers!

Anonymous Functions

- “`auto f1 = [] (double x) {return x * x;};`” defines “f1” (a variable) and binds it to a function
 - Function as a *storable* value (can be assigned to a variable)
 - Function as an *expressible* value (can be the result of an expression)
- “[] (. .) { . . . }” defines a *function without a name!!!*
- This expression (named “lambda” in C++) evaluates to an *anonymous function*
 - Can be assigned to a variable, passed as an argument to a function, ...
- The type of a lambda expression in C++ is “`std::function<...>`”

Formal Arguments and Actual Arguments

- If $f(x) = x + 1$, applying “f” to “2” requires to:
 - Replace “f” (function name) with “ $x + 1$ ” (function body)
 - Replace “x” (formal parameter) with “2” (actual parameter)
 - Compute the result $2 + 1 = 3$
- In C-like languages, we are used to look at function invocation in a different way:
 - Push “2” (actual parameter) on the stack
 - Call the function body (which pulls the parameter’s from the stack)
 - Different argument-passing methods

Passing Parameters by Value

- Only possible method in C
- One local variable is allocated (on the stack) when the function is called
- The local environment contains a binding between the formal parameter's name and this local variable
- The variable is automagically initialized with the value of the actual parameter

```
int f(int n)
{
    n = n + 1;

    return n * 2;
}
```

Passing Parameters by Reference

- Possible in C++
- No local variable for the formal parameter
- The local environment contains a binding between the formal parameter's name and the actual parameter
 - The actual parameter **must** be an L-Value
 - The formal parameter is an alias for the actual parameter

```
int f(int &n)
{
    n = n + 1;

    return n * 2;
}
```

Passing Pointers by Value

- “Emulation” of reference passing in C
- A pointer to the “real” actual parameter is passed by value
- First difference with parameter passing by reference: syntax
 - But there are other notable differences... For example, in this case the formal parameter is still a local variable!
 - Think about “ $n = n + 1$ ” in the example below

```
int f(int *n)
{
    *n = *n + 1;

    return *n * 2;
}
```

Passing Parameters by Name

- Seen for functional programs evaluation
 - Function name replaced by function body
 - Formal parameter replaced by actual parameter
- Not very useful for imperative languages...
 - Parameters can be evaluated every time they are used... Think about “ $x + x$ ” with actual parameter “ $i++$ ”!
- ...But good model for how FP reduction works!

Issues with Parameters by Name

```
int a = 1;

int f(int v)
{
    int a = 666;

    return a + v;
}
```

- What is $f(a)$ if the parameter is passed by name?
- { `int a = 666; return a + a;` }... Returns 1332!
- If the name of the local variable is changed to “b”, we get { `int b = 666; return b + a;` } and the return value is 667!
- The return value depends on the name of a local variable???

Call by Name, Again

- consider this code:

```
int infinite_recursion(int z)
{
    return infinite_recursion(z);
}
```

```
int select(int n, int x, int y)
{
    return n == 0 ? x : y;
}
```

- What happens in C++ (parameters passed by value) when calling
`select(0, 1, infinite_recursion(1))`?
- What would happen if parameters were passed by name?
 - Can you emulate pass-by-name, in this case?

Evaluation, or... Reduction

- In the FP jargon the term “reduction” is often used instead of “evaluation”
 - A program is **reduced** by text replacement of subexpressions
- Substituting function invocations with the function body, and then with the returned values
 - Substitute the formal parameter with the actual parameter...
 - For example, if
`double f(double x) {return x * x; }`, we want
“`f(3)`” to be replaced by “`3 * 3`” and then “`9`”
- Let’s look at some more details about how reduction works...

Reduction?

- Function application:
 - Replacement of the function name with the function body
 - Replacement of formal parameters with actual parameters
- Often called parameters passing **by name**
- Example: in “ $f(3)$ ”, “ f ” is first replaced by “ $x * x$ ” and then “ x ” is replaced by “ 3 ” obtaining “ $3 * 3$ ”, which evaluates to “ 9 ”
 - $f(3) \rightarrow (x * x)(3) \rightarrow 3 * 3 \rightarrow 9$
- It is all strings manipulation!
 - No variables, no execution, no stack...

Example of Reduction

```
unsigned int fac(unsigned int n)
{
    return n == 0 ? 1 : n * fac(n - 1);
}
```

- `fac(4)` is replaced by
“`n == 0 ? 1 : n * fac(n - 1)`” applied to “4”...
 - Replacement due to the definition of “`fac()`”
- Then, “`n`” is replaced by 4
 - Replacement due to parameters passing
- “`4 == 0 ? 1 : 4 * fac(4 - 1)`” evaluates to
“`4 * fac(3)`”
 - Replacement due to mathematical evaluation!
- Now, restart from the beginning with “`fac(3)`”...

Diverging Computations

- It is possible to create endless sequences of replacements
 - `int f(int x) {return f(x);}`
 - This is equivalent to an endless loop (`“while(1);”`): diverging computation
- In other words, an infinite recursion is a diverging computation
 - Will the stack overflow? Not if we use tail calls (and corresponding optimizations)
- Looks strange, but is needed for Turing completeness!!!

Functional Programming Concepts

- Repeat with me: **no commands** (no side effects), only use **expressions** (pure functions)
 - Expressions are composed by values (non-reducible) and primitive operators
- How are expression built? (what's the syntax for writing expressions?)
- Two basic concepts: **abstraction** and **application**
 - In few words, “abstraction” is function definition...
 - ...While “application” is function application
- Text replacements are performed based on abstractions and applications
 - Text replacements due to mathematical evaluation can be seen as a form of “application”

Abstractions

- Abstraction: given an expression “ e ” and an identifier “ x ”, builds an expression returning a function that has “ e ” as body and “ x ” as formal parameter
 - The expression “ e ” can then use the variable “ x ”
- In FP jargon, we are *abstracting* e from the specific value of x
- Example of abstraction: `[] (auto x) e`
 - Anonymous function mapping x into e !!!

Applications

- Application: given a function f and an expression e , builds the expression $f(e)$
 - Applies f to e , evaluating the value of $f()$ given the value of e
 - This is the inverse of abstraction!

Reduction Revisited

- The reduction of an expression happens using 2 fundamental mechanisms:
 1. Search in the environment (replacing identifiers with the corresponding values)
 2. Function application (replacing formal parameters with actual parameters)
- Replacing “`fact (4)`” with the function body is based on a search in the environment (search the environment for the value corresponding to symbol “`fact`”)
- Replacing “`n == 0 ? 1 : n * fact (n - 1)`” with “`4 == 0 ? 1 : 4 * fact (4 - 1)`” is based on function application

Summing Up: Functional Languages Features

- Functions are expressible values
 - Functions (code) and data are handled in the same way
- Functions can receive functions as arguments
- Functions can generate functions as results
 - Looks simple, but...
 - What's the environment of the returned function?
We need **closures**!
- People often talk about *high-order functions*...

Putting all Together

- A functional program is a set of definitions and expressions
 - Can modify the environment (creating bindings)
 - Can require the evaluation of complex functions
- Executed by text replacement (reduction)
- Continuously simplify expressions using 2 operations:
 - Search (bindings in the environment) and replace
 - Applications of functions to arguments (replacing formal parameters with actual parameters)

Some Questions...

- This “search and replace” (and apply) idea looks simple
 - But the devil is in the details!
- When should the reduction process stop?
 - What is an “irreducible expression” (or, value)?
- If more than 1 replacement can be performed in the same expression, which one is performed first?
 - What is the “precedence rule” for replacements/reductions?