Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

Foundations of Functional Programming

Francisco Ortín Soler

University of Oviedo

Code Examples

- All the example **code** shown in these slides is **available** for download
  - The directory is shown in underlined blue font (such as delegates/lambda)
- To understand the concepts explained, the source code must be **opened**, **analyzed**, **modified**, **executed** by the students, making sure that they **understand** the code

Content

- Lambda Calculus
- Curry-Howard Isomorphism
- First-Class Functions
- Closures
- Currying
- Partial Application
- Continuations
- Lazy Evaluation
- Referential Transparency
- Pattern Matching
- Higher-Order Functions
- List Comprehensions

Functional Paradigm

- **Declarative** paradigm that abstracts programs as mathematical **functions** computing **immutable data**
  - Data is never modified
  - Instead of modifying data, a function is called returning the new data without modifying the original one
- A **program** is defined as a set of functions invoking one another
- Opposite to imperative programming, functions can **never** have **side effects**:
  - the value of an function invocation only depends on the value of its parameters
  - always returning the same value for the same values of the parameters (pure functional paradigm)

Lambda Calculus

- Functional programming has its roots in **lambda calculus** defined by *Alonzo Church* and *Stephen Kleene* in the 30s
- Lambda calculus (λ-calculus) is a formal system based on **function definition** (abstraction) and its **application** (invocation)
  - Makes extensive use of recursion
- Lambda calculus is considered as the **smallest universal programming language**
  - **Universal** in the sense that any computable function can be expressed using this formalism (a.k.a., Turing complete), i.e. the Church-Turing thesis
- Therefore, it is widely used for language researchers and developers

Lambda Expressions

- A **lambda expression** (or term) is defined as
  - A lambda **abstraction** $\lambda x.M$ (M, N, M_1, M_2…)
    where x is a variable (x, y, z, x_1, x_2…) *–parameter–*
    and M is a lambda expression *–function body*
  - An **application** $M\ N$
    where both M and N are lambda expressions
- Examples of **abstractions** (functions)
  - The identity function `f(x)=x` can be represented with the expression $\lambda x.x$
  - The double function `g(x)=x+x` can be represented with the expression $\lambda x.x+x$

    Note: *The `x+x` is not a lambda expression actually. It can be represented with a (longer) lambda expression (obviated for the sake of brevity)*

Application (β-reduction)

- Function **application** means function **invocation**
- Function application is defined the following way

  $(\lambda x.M)N \rightarrow M[x:=N]$ (or $M[N/x]$)

  Being *x* a variable and

  M and *N* lambda expressions

  And *M[x:=N]* (or *M[N/x]*) represents *M* with all the occurrences of *x* are substituted by *N* (substitution)
- This substitution (expression or term reduction) is called **β-reduction**
- Examples of function application
  - $(\lambda\underline{x}.\underline{x}+\underline{x})3 \rightarrow 3+3$
  - $(\lambda\underline{x}.\underline{x})\ (\lambda y.y*2) \rightarrow \lambda y.y*2$

Church-Rosser Theorem

- In some lambda terms, two or more β-reductions may be applied
  - (λx.x) (λy.y*2) 3
- The **Church-Rosser theorem** states that the ordering in which the reductions are chosen does not make a difference to the eventual result:
  - (λx.x) (λy.y*2) 3 → (λy.y*2) 3 → 3*2
  - (λx.x) (λy.y*2) 3 → (λx.x) (3*2) → 3*2
- Therefore, parenthesis are mainly used to delimit lambda terms:
  - (λx.x) λy.y is a function application, evaluated reduced to λy.y
  - λx.x λy.y is just one function abstraction

http://www.cburch.com/proj/lambda (uses prefix syntax for arithmetic expressions and \ for the λ symbol)

Bound and Free Variables

- In the following abstraction $\lambda x.xy$
  - The *x* variable is **bound**
  - The *y* variable is **free**
- Upon substitution, only free variables are substituted

  $(\lambda x.x(\lambda x.2+x)y)M \rightarrow x(\lambda x.2+x)y\ [x:=M] = M(\lambda x.2+x)y$
  - Notice that the second *x* is not substituted (it is bound) because it represents a different variable; with the same name, though
- To avoid this name conflict, the **α-conversion** can be applied:

  *All the occurrences of a **bound** variable in an abstraction can be renamed with a new **fresh** variable*

  $(\lambda x.x(\lambda \underline{x}.2+\underline{x})y)M \equiv (\lambda \underline{x}.\underline{x}(\lambda \mathbf{z}.2+\mathbf{z})y)M \rightarrow$
  $\underline{x}(\lambda z.2+z)y\ [x:=M] = M(\lambda \underline{z}.2+\underline{z})y \equiv M(\lambda x.2+x)y$

α-conversions

- Which of the following couples of lambda terms are **α-equivalent**?

λx.xy	λz.zy
λx.2+x	λy.2+y
xy(λx.2+x)	xy(λz.2+z)
λx.xy	λy.yy

α-Conversion

- Due to α-conversions, functions may be applied to themselves
- **Example 1**. Applying the identity function to itself gets another identity function

  $(\lambda x.x)(\lambda x.x) \equiv (\lambda x.x)(\lambda y.y) \rightarrow \lambda y.y \equiv \lambda x.x$

  Apply α-conversions to avoid different bound variables to have the same name before function application
- **Example 2**. We define the *double* function ($\lambda x.x+x$) and the *apply twice* function ($\lambda f.\lambda x.f(fx)$), computing *twice the double of n*

α-Conversion

- By convention expressions are evaluated from left to right

$$(\lambda f.\lambda x.f(fx))(\lambda x.x+x)n \equiv (\lambda f.\lambda x.f(fx))(\lambda y.y+y)n \rightarrow$$

α-conversion on 2nd expression (x/y)

$$(\lambda x.(\lambda y.y+y)((\lambda y.y+y)x))n \equiv (\lambda x.(\lambda y.y+y)((\lambda z.z+z)x))n$$

β-reduction [f:=(λy.y+y)] | α-conversion on 1st expression (y/z)

$$\rightarrow (\lambda y.y+y)((\lambda z.z+z)n) \rightarrow ((\lambda z.z+z)n) + ((\lambda z.z+z)n) \equiv$$

β-reduction [x:=n] | β-reduction [y:=(λz.z+z)n]

$$((\lambda z.z+z)n) + ((\lambda x.x+x)n) \rightarrow (n+n) + ((\lambda x.x+x)n)$$

α-conversion on 2nd expression (z/x) | β-reduction [z:=n]

$$\rightarrow (n+n)+(n+n)$$

β-reduction [x:=n]

Seminar 3

- Seminar 3
  - Lambda Calculus

Halting Problem

- **Halting Problem**: Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever
- In 1936 Alan Turing proved that it is **impossible** to write a general **algorithm** to solve the halting problem **for all possible program-input pairs** of a Turing-complete language ⇒ ***undecidable*** problem
  - A Turing-complete language is that capable of simulating the Turing machine (type 0 or recursively enumerable language)
- There are languages such as Coq and Adga that allow ensuring that **a** program terminates
- There exist lambda expressions that continue to run forever (**do not converge**)

  $(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$

Logic as the basis for Software

- **Logic** is the mathematical basis for **software engineering**

$$\frac{\text{Logic}}{\text{Software Engineering}} = \frac{\text{Calculus}}{\text{Civil Engineering}}$$

- **Logic** is the formal science of studying modes of valid reasoning and its proof
  - Study of the methods and principles to distinguish correct from incorrect reasoning
- A **proof** is a demonstration that a **proposition** (**theorem**) is necessarily true
  - Proofs are obtained from **deductive reasoning**
- The different proofs of a proposition (theorem) are called **evidences**

Curry-Howard

- **Curry-Howard** isomorphism (or correspondence) establishes a direct relationship between programs and proofs
  - Thus, establishing a correspondence between **logic** and **computation**
  - In lambda calculus the correspondence is direct
- It involves that
  1. There is a correspondence between **types** and **propositions** (theorems or formulas)

Logic	Equivalent to	Type
$\supset$ ($\rightarrow$)		Function
$\wedge$		Product type (tuple or register)
$\vee$		Sum type (union)
true		Top type (Object)
false		Bottom type (void)
$\forall$		Generics

Curry-Howard, Types

2. There is a correspondence between **programs** and **evidences that proofs** the **propositions** (theorems or formulas) **described by the program's type**

   Example: $A \wedge B \supset B \wedge A$ is a true proposition (theorem or formula) that can be **proved** by means of natural deduction the following way:

Inference Rule:

$$\frac{\text{Premise}}{\text{Conclusion}}$$

$$\dfrac{\dfrac{A \wedge B}{B} \qquad \dfrac{A \wedge B}{A}}{\dfrac{B \wedge A}{A \wedge B \supset B \wedge A}}$$

Curry-Howard, Types

Example (continued): The **proposition** (theorem) $A \wedge B \supset B \wedge A$ corresponds to the **type** $A \times B \rightarrow B \times A$ and the previous **proof** corresponds to inferring the type of the **program**:

$$\dfrac{\dfrac{\texttt{x} : A \times B}{\texttt{second x} : B} \qquad \dfrac{\texttt{x} : A \times B}{\texttt{first x} : A}}{\texttt{<second x,first x>} : B \times A}$$

$$\dfrac{\texttt{<second x,first x>} : B \times A}{\lambda \texttt{x}:A \times B.\texttt{<second x,first x>} : A \times B \rightarrow B \times A}$$

x : T

Means "*x has the T type*"

Curry-Howard Correspondence

- The deductive process used is the same for:
  - *1)* Proving the proposition (theorem) $A \wedge B \supset B \wedge A$
  - *2)* Finding the program which type is $A \times B \rightarrow B \times A$

1)

$$\dfrac{\dfrac{A \wedge B}{B} \qquad \dfrac{A \wedge B}{A}}{\dfrac{B \wedge A}{A \wedge B \supset B \wedge A}}$$

2)

$$\dfrac{\dfrac{\texttt{x} : A \times B}{\texttt{second x} : B} \qquad \dfrac{\texttt{x} : A \times B}{\texttt{first x} : A}}{\dfrac{\texttt{<second x,first x>} : B \times A}{\lambda \texttt{x:A} \times \texttt{B.<second x,first x>} : A \times B \rightarrow B \times A}}$$

Proving Program Properties

- The consequence is that **logic** can be used as a **formal mechanism** to **prove software properties**

$$\frac{\text{Logic}}{\text{Software Engineering}} = \frac{\text{Calculus}}{\text{Civil Engineering}}$$

- The best example is **program verification**: assure that programs fully satisfy all the expected requirements
  - For instance, a **sorting algorithm** is correct when the following is proved to be true
  1. That, after the invocation, all the elements in the returned collection are sorted,
  2. **For any possible collection** (infinite) passed as a parameter

Logic and Functional Paradigm

- The **functional paradigm** is the one **most used** to **prove program properties** because
  1. Every computation can be expressed in lambda calculus (it is universal)
  2. There is a direct translation to logic (Curry-Howard isomorphism)
- There exist **proof assistants** to allow proving program properties (Coq, Isabelle, Agda, Twelf…)
  - Are based on functional languages (ML and Haskell)
  - Include a language to prove program properties by means of natural deduction (proofs are programs!)
  - Allow program extraction: code generation for different languages (OCaml, Haskell, Scala, JavaScript…) after proving the program is correct

Functions, First-Class Values

- The functional paradigm identifies functions as first-class values, just like the rest of values (e.g., built-in or objects): **first-class functions**
- This means that functions are **another common type**, allowing variable instantiation to:
  - Store them in data structures (objects, structs, trees, tuples, graphs…)
  - Pass them as arguments in a method call
  - Return them as the value of a method call
- A function is said to be a **higher-order function** if
  - Either receives a function as a parameter
  - Or returns a function as a result

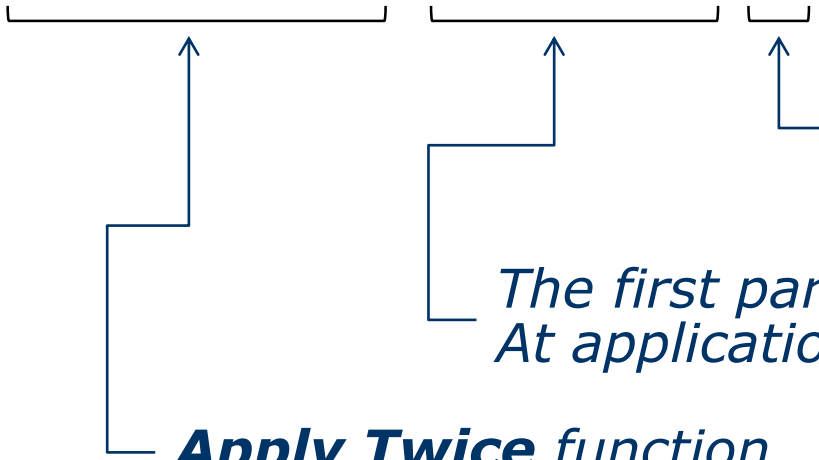
Higher-Order Functions

- The **apply twice** function previously shown is a higher-order function

  $\lambda f.\lambda x.f(fx)$

- Receives a function as a parameter (**f**) and another expression (**x**). Passes **x** to **f** (**fx**) and the returned value is passed another time to **f** (**f(fx)**)

$(\lambda f.\lambda x.f(fx))\quad(\lambda y.y+y)\quad 3 \rightarrow 12$



Delegates

- C# provides first-class functions (methods) by means of **delegates** (and lambda expressions)
- A delegate is a C# **type representing an** instance or class (`static`) **method**
- Delegate **instances** (variables) represent method references
- Therefore,
  - Passing delegates as parameters imply passing methods (functions) to another method (function)
  - Delegate assignment allows parameterizing data structures (and algorithms) with functions (behavior)

Delegates

- The following line defines **the delegate type**

```
public delegate int Comparison(Person p1, Person p2);
```

- **Comparison** is a method type that receives two **Person** instances and returns an **int**
- It can be used to declare the following **SortPeople** method, which sorts objects with any comparison criterion

```
static public  void SortPeople(
            Person[] vector, Comparison comparison) {
```

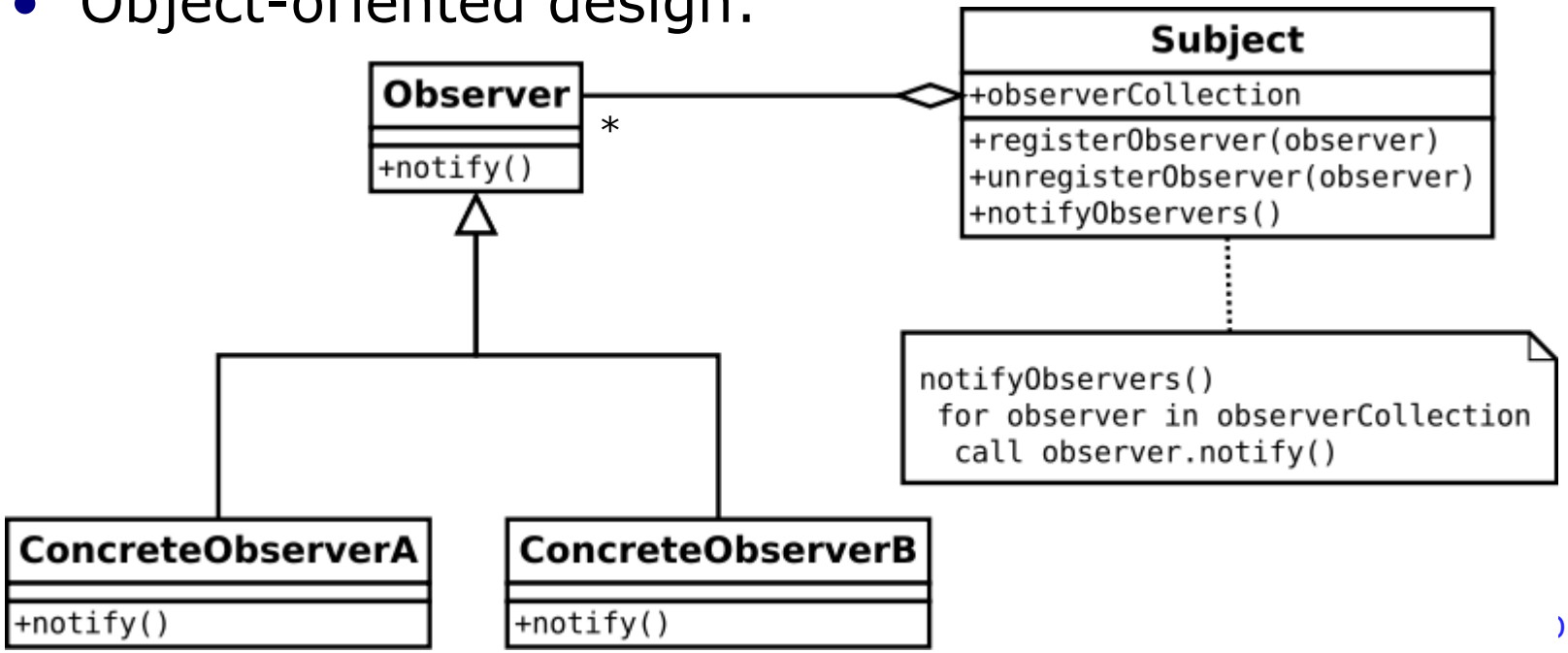
- And use the delegate variable the following way

```
if (comparison(vector[i], vector[j]) >0) {
  Person temp = vector[i];
  vector[i] = vector[j];
  vector[j] = temp;
}
```

- delegates/delegates

The *Observer* Design Pattern

- An widespread use of delegates in .NET is the **Observer** design pattern
  - The subscribers (`Observer` or `Listener`) (un)register themselves in a `Subject`
  - When an event is triggered, the `Subject` notifies the registered `Observer`s of the event
- Object-oriented design:



The *Observer* Design Pattern

- **A delegate** instance can actually collect a set of methods
  - The **+=** operator adds (registers) a new method to the collection
  - The **-=** operator removes (unregisters) an existing method form the collection
  - The **=** operator updates the whole collection with one single method
- When the delegate is called, all the registered methods will be invoked
- This is how delegates are used to implement the Observer design pattern

The *Observer* Design Pattern

```
class Button {
  public delegate void ButtonClick();
  private ButtonClick methodsToBeCalled;
  public void AddMethod(ButtonClick method) {
    methodsToBeCalled += method;
  }
  public void RemoveMethod(ButtonClick method) {
    methodsToBeCalled -= method;
  }
  public void Click() {
    methodsToBeCalled();
  }
}
```

The *Observer* Design Pattern

```
static void Main() {
  Button button = new Button();

  Integer integer = new Integer(3);
  Controller controller = new Controller();

  button.AddMethod(integer.Show);
  button.AddMethod(controller.OnClick);

  button.Click();
}
```

- delegates/observer

Generic Delegate Types

- The .NET platform offers a set of useful delegates
  - They utilize generics
- The most common ones are **`Func`**, **`Action`** and **`Predicate`**
  - **`Func<T>:`** Parameterless method that returns **`T`**
  - **`Func<T1,T2>`**: Method with one **`T1`** parameter that returns **`T2`**
  - …
  - **`Action`**: Parameterless method that returns no value
  - **`Action<T>`**: Method with one **`T`** parameter that returns no value
  - …
  - **`Predicate<T>`**: Method returning a **`bool`** and receiving a **`T`** parameter
- This predicates are really useful (**do not define new ones**!)
- delegates/predefined

Anonymous Delegates

- Implementing a method for just passing it as a parameter is tedious
- In the functional paradigm it is common to write a function at invocation, when passed as a parameter
- The first approach C# provided for this purpose was **anonymous delegates**
  - A syntax to define a delegate instance writing the method parameters and its body

```
Person[] people = PersonPrintout.CreatePeopleRandomly();
Person[] beyond18 = Array.FindAll(people,
                delegate(Person p) { return p.Age >= 18; }
                );
```

  - delegates/anonymous.methods
  - The **syntax** is quite **verbose**

Lambda Expressions

- C# lambda expressions allow writing **complete functions as expressions** (following the λ-calculus approach)
  - It is an improvement of anonymous delegates
- **Syntax**
  - Parameters (if any) are separated by commas
    - Parameter types can optionally be declared
    - Brackets must be used for more than one parameter
    - Zero parameters are declared with `()`
  - The `=>` token separates parameters from the function body
  - If the body has more than one sentence, they are delimited with `;`
  - If the function body is a single expression, the `return` keyword and `{}` are optional

Lambda Expressions

- Types of lambda expressions promote to the standard .NET delegate types: **`Func`**, **`Predicate`** and **`Action`**

```
Func<Func<int, int>, int, int> applyTwice =
                              (f, n) => f(f(n));
Console.WriteLine(applyTwice( n => n+n, 3));

Person[] people = PersonPrintout.
                          CreatePeopleRandomly();
Person[] above18 = Array.FindAll(
          people, person => person.Age >= 18);
```

- delegates/lambda

Lambda Expressions

- More examples:

```
IDictionary <string, Func<int, int, int>> functionalCalc = new
Dictionary<string, Func<int, int, int>>();
functionalCalc[“add"] = (op1, op2) => op1 + op2;
functionalCalc[“sub"] = (op1, op2) => op1 - op2;
functionalCalc[“mul"] = (op1, op2) => op1 * op2;
functionalCalc[“div"] = (op1, op2) => op1 / op2;

functionalCalc[“add"](3, 4); // 7

IList<Predicate<string>> tests = new List<Predicate<string>>();
tests.Add(s => s.Length < 5);
tests.Add(s => !s.StartsWith(“F"));
tests.Add(s => s.EndsWith(“i"));

string str = “Hi";
foreach (var test in tests) { if (!test(str)) {…} }
```

Seminar 4

- Seminar 4
  - First-Class Functions

Iteration and Recursion

- Pure functional programming **does not support iterations**
  - **Recursion** is provided instead
- In C# (and most languages) functions have names, so it is easy to perform a recursive call

```
static ulong fact(ulong n) {
    return n==0||n==1 ? 1 : n*fact(n-1);
}
```

- However, in lambda calculus functions do no have names, making it difficult to implement recursion

```
λx.if x=0 or x=1 then 1 else x*?(x-1)
```

The `if-then-else`, `=`, `*` and `-` operators can also be coded in lambda calculus

- How do we solve this?

Iteration and Recursion

- Recursive functions should be implemented as higher-order functions, receiving themselves as a parameter

```
λf.λx. if x=0 or x=1 then 1 else x*(f x-1)
```

- The question is, **how can the function itself be passed as a parameter?**

Fixed-Point Combinator

- A **Fixed-point combinator** (commonly named `fix`) is a higher-order function that satisfies that

  `fix f` $\rightarrow$ `f (fix f)`

- In other words, applying `fix` to `f` (i.e., `fix f`)
  - Returns `f`
  - Passing a new invocation to `fix f` as the first parameter (which is `f`)

Example Invocation

- Recall

a) `fact ≡ λf.λx.if x=0 or x=1 then 1 else x*(f x-1)`

b) `fix fact → fact (fix fact)`

- Let's evaluate `fix fact 3` (evaluated sub-expressions are underlined)

1.	`fix fact 3`	b) →
2.	`fact (fix fact) 3`	a) →
3.	`3 * (fix fact 2)`	b) →
4.	`3 * (fact (fix fact) 2)`	a) →
5.	`3 * 2 * (fix fact 1)`	b) →
6.	`3 * 2 * (fact (fix fact) 1)`	a) →
7.	`3 * 2 * 1`	

Y Combinator

- A widely known `fix` function is the **Y Combinator**

$$fix \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$\boxed{fix\ fact \equiv}$$

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))\ fact \rightarrow$$

$$\boxed{(\lambda x.fact(xx))(\lambda x.fact(xx))} \equiv_{\alpha}$$

$$(\lambda y.fact(yy))(\lambda x.fact(xx)) \rightarrow$$

$$fact\ (\boxed{(\lambda x.fact(xx))(\lambda x.fact(xx))}) \equiv$$

$$fact\ (fix\ fact)$$

Closures

- A **closure** is a first-class function together with its referencing environment: a table storing a reference to each free variable
- In lambda calculus, λy. … **x** is a closure in

  λx. … x… λy. … **x**

  x is a free variable in λy. … **x** but a **reference** (not a copy) to the outer *x* in λx. … x… is stored
- In C#

```
int value = 1;
Func<int> doubleValue = () => value * 2;
doubleValue();  // 2
value = 7;
doubleValue();  // 14
```

Closures

- The free variables in a closure represent **state**
  - This **state** can even be **hidden** when the variable environment (scope) has been released

```
static Func<int> ReturnCounter() {
  int counter = 0;
  return () => ++counter;
}
```

- Therefore, closures may represent **objects** (encapsulation)

Closures

- Closures can also represent **control structures**

```
Void WhileLoop(Func<bool> condition, Action body) {
  if (condition()) {
    body();
    WhileLoop(condition, body);
  }
}
…
int i = 0;
WhileLoop(   () => i < 10,
             () => { Console.Write(i); i++; }
          );
```

- closures/closures

Closures

- Closures allow representing
  1. Iterative control structures
  2. Objects and information hiding
- Therefore, we can see how this imperative features can be translated into equivalent elements of the functional paradigm
  - There is no loss of expressiveness
  - But a **different way** (paradigm) of **expressing the abstractions**

Currying

- Lambda calculus is a **universal** programming language: any computable function can be expressed using this formalism
  - Numbers and integer and Boolean operations (*Church encoding*)
- However, a lambda abstraction is defined with a single parameter
- **Currying** is the technique of transforming a function with multiple parameters in a function with a single parameter
  - The function receives a parameter and returns another function that can be called with the second parameter
  - This technique is repeated for all the parameters
  - The invocation is a chain of function calls (i.e., `f(1)(2)(3)` vs. `f(1, 2, 3)`)
- The name is a reference to the mathematician Haskell Curry (invented by Moses Schönfinkel)

Currying

- In lambda calculus

  $\lambda x.\lambda y.x+y \equiv \lambda xy.x+y$

- In C#

```
static Func<int, int> CurriedAdd(int a) {
  return b => a + b;
}
static Predicate<int> GreaterThan(int a) {
  return b => a > b;
}
…
CurriedAdd(3)(1)
Predicate<int> isNegative = GreaterThan(0);
```

- Question: What is the feature used to obtain currying?
- Its main benefit is **partial application** (following slides)
- closures/currying

Partial Application

- When **functions** are **curried**, it is possible to partially apply (invoke) them
  - In Haskell and ML, all the functions are curried
- **Partial application** means passing a lower number of arguments to the function application (invocation)

  obtaining another function with a smaller *arity* (number of parameters)
- This provides a powerful feature, especially **when operators are also curried** (e.g., Haskell)
- In lambda calculus

```
(λx.λy.x+y)1 ≡  λy.1+y
(add)1       ≡  increment
```

Partial Application in C#

- Partial application **syntax** in C# is **verbose**
  1. Functions (methods) are not curried
  2. Operators are not functions (not curried either)
- This implies that partial application in C# is not as useful as in Haskell (which provides both features)
- A (verbose) example in C#:

```
static Func<int, int> Add(int a) {
  return b => a + b;
}
static Predicate<int> GreaterThan(int a) {
  return b => a > b;
}
static Predicate<int> SecondParameter(
        Func<int, Predicate<int>> predicate, int parameter) {
  return a => predicate(a)(parameter);
}
```

Partial Application in C#

```
static int[] Map(this int[] source, Func<int, int> function) {
  int[] destination = new int[source.Length];
  for (int i = 0; i < source.Length; i++)
    destination[i] = function(source[i]);
  return destination;
}
static IEnumerable<int> Filter(this IEnumerable<int> source,
                               Predicate<int> predicate) {
  IList<int> destination = new List<int>();
  foreach (int item in source)
    if (predicate(item)) destination.Add(item);
  return destination;
}
static void Main() {
  int[] integers = new int[10];
  int[] inc = integers.Map( Add(1) );
  var positive = integers.Filter( SecondParameter(GreaterThan, 0) );
}
```

- closures/partial.application

(1+) in Haskell

(>0) in Haskell

Continuations

- A **continuation** is the representation of state of computation at a given point of execution
- The state of computation at least consists of
  1. The **state of the runtime stack**
  2. The **next instruction to be executed**
- Those languages that provide continuations allow storing their state of execution and restore it later
  - Scheme and Ruby provide continuations
  - C# does not provide first-class continuations (it provides generators with `yield`)

Continuations

- Languages that support continuations provide the **`call/cc`** (*call with current continuation*) function to obtain the state of execution
- The following is an example using the C# syntax (**not valid in C#**)

```
static Func<int> continuation = null;
static int Test() {
  int i = 0;
  // Assigns to continuation the dynamic state of the program
  // (in this line of execution)
  callcc(state => continuation = state);
  i = i + 1;
  return i;
}
static void Main() {
  Console.Write(Test());          // 1
  Console.Write(continuation());  // 2
  Console.Write(continuation());  // 3
  Func<int> previousContinuation = continuation;
  // Resets the continuation
  Console.Write(Test());          // 1
  Console.Write(continuation());  // 2
  // Uses the first one, once again
  Console.Write(previousContinuation());  // 4
}
```

Generators

- A **generator**
  - is a function (method)
  - that simulates returning a collection of elements
  - without building and filling the whole collection
  - returning one element each time it is called
- Generators can be implemented with **continuations**
- This approach is **more efficient** because
  - Requires less memory
  - The first values are processed and returned immediately
  - Only those elements that are used need to be generated
- Therefore, a generator is a function that **behaves like a *lazy* iterator**

Generators in C#

- C# provides generators by means of **`yield`**, a sort of *ad hoc* continuation

```
static IEnumerable<int> InfiniteFibonacci() {
  int first = 1, second = 1;
  while (true) {
    yield return first;
    int addition = first + second;
    first = second;
    second = addition;
} }
...
foreach (int value in Fibonacci.InfiniteFibonacci()) {
  Console.WriteLine("Term {0}: {1}.", i, value);
  if (i++ == numberOfTerms) break;
```

Generators in C#

```
IEnumerable<int> FiniteFibonacci(int maximumTerm) {
  int first = 1, second = 1, term = 1;
  while (true) {
    yield return first;
    int addition = first + second;
    first = second;
    second = addition;
    if (term++ == maximumTerm)
      yield break;
  }
}
...
foreach (int value in Fibonacci.FiniteFibonacci(10))
  Console.Write(value);
```

- continuations/generators

Lazy Evaluation

- **Lazy evaluation** (a.k.a. call by need) is an evaluation strategy that delays the evaluation of an expression until its value is needed
  - The opposite is **eager** or **strict** evaluation
- Most languages provide the eager evaluation of argument values
  - Haskell and Miranda provide lazy argument evaluation
  - An example in C# syntax (**non-valid C# code**)

```
int Eager(int n) { return 0; }
int Lazy(int n)  { return 0; }
int a = 1, b = 1;
Eager(a++); // a == 2 after invocation
Lazy(b++);  // b == 1 after invocation
```

Lazy Evaluation

- The benefits of lazy evaluation are
  1. Lower **memory consumption**
  2. Higher **runtime performance**
  3. The capability of creating **potentially infinite data structures** and **algorithms**
- The implementation of a language processor (compiler or interpreter) that provides lazy evaluation is **more complex**

Lazy Evaluation in C#

- C# **does not provide lazy evaluation**
  - Except for the continuations implemented to support generators
- Since the generation of elements is lazy, collections of infinite elements can be generated with **`yield`**

  and used it together with the two following extension methods
  - **`Skip`** bypasses a set of elements in a sequence, returning the remaining sequence
  - **`Take`** to return a specified number of contiguous elements from the start of a sequence

Lazy Evaluation in C#

```
static private IEnumerable<int> LazyPrimeNumbersGenerator() {
  int n = 1;
  while (true) {
    if (IsPrime(n))
      yield return n;
    n++;
  }
}
static internal IEnumerable<int> LazyPrimeNumbers(
                                  int from, int numberOfNumbers) {
  return LazyPrimeNumbersGenerator()
                      .Skip(from).Take(numberOfNumbers);
}
```

- continuations/lazy

Referential Transparency

- An expression is **referential transparent** if it can be replaced by its value without changing the behavior of the program
  - The opposite is referential opaqueness
- While in mathematics all functions are referentially transparent, in programming this is not always the case
- Referential transparency is one of the **principles of functional programming**
- When offered by a programming language, it is said to be **pure functional**: Haskell, Clean and Charity
- Elements that make a language not to be referential transparent are
  1. Global mutable variables
  2. (Destructive) assignments (**=**, **+=**, ***=**, **++**, **--**…)
  3. Impure functions (I/O, *random*, *datetime*…)

Referential Transparency

- Question: If a language provides closures, can it also provide referential transparency?

Global Variables & Assignments

1. **Mutable variables outside the function scope** avoid referential transparency
   - The `ReturnCounter` closure returns an integer whose value depends on the number of previous invocations ⇒ It cannot be replaced by the value of a previous invocation

```
static Func<int> ReturnCounter() {
  int counter = 0;
  return () => ++counter;
}
```

2. It happens the same with (destructive) **assignments**
   - Values of variables depend on their previous assigned values

Pure Functions

3. The use of **non-pure functions** implies **referential opaqueness**
   - A function is **pure** when
     1. Always evaluates the same result value given the same argument values

        Does not depend on any dynamic mutable state or on an input / output device
     2. Its evaluation does not cause any side effect or output

        Global variables are not modified, and no input / output devices are accessed either
   - Examples of **non pure** functions: `DateTime::Now`, `Random::Random`, `Console::ReadLine`
   - Examples of **pure** functions : `Math::Sin`, `String::Length`, `DateTime::ToString`

Benefits

- The benefits of **referential transparency** are:
  1. **Mathematical reasoning can be applied to programs**, so that:
     - Its **correctness** can be proved ⇒ Demonstrating that a program is correct with respect to a specification (computing a value, fulfilling invariants and postconditions…)
     - **Reasoning** about its **semantics** ⇒ Program properties can be worked out (termination, runtime errors, memory consumption…)
  2. Programs can be **transformed**, so that:
     - They can be **simplified** and **parallelized**
     - They can be **optimized** ⇒ memoization, common subexpression elimination, parallelization…

Memoization

- As an example, we will see an **optimization technique** that can be applied to **referential transparent expressions**: **Memoization**
  - A referential transparent expression can be substituted with its value
  - If the expression is a function invocation, it can be substituted by its return value
  - The first time the function is called, its return value can be stored in a cache
  - In the subsequent invocations, the value is retrieved from the cache, and hence the function is not evaluated

Memoization

```
static class MemoizedFibonacci {

  private static IDictionary<int, int> values =
                    new Dictionary<int, int>();

  internal static int Fibonacci(int n) {
    if (values.Keys.Contains(n))
      return values[n];
    int value =  n <= 2 ? 1 :
                 Fibonacci(n - 2) + Fibonacci(n - 1);
    values.Add(n, value);
    return value;
  }
}
```

- Questions:
  - Advantages?
  - Disadvantages?

- continuations/memoization

Lazy Evaluation (Revisited)

- Recall that **the lazy parameter** passing mechanism
  - delays the evaluation of a the parameter until its value is needed
- This behavior can be achieved by
  - Coding the **parameters as functions** (functions will then be higher-order)
  - Memoizing (saving) its **evaluation**
- Mandatory Activity: Analyze, modify, execute and comprehend the following C# project
  - continuations/lazy.simulated

Pattern Matching

- **Pattern matching** is the act of checking a sequence of **elements** for the presence of constituents of some **pattern**
- The **elements** commonly are
  - Types (classes) of variables (objects)
  - Lists
  - Strings
  - Tuples
  - Arrays
  - …
- The **patterns** commonly are
  - Sequences (generally expressed by regular expressions)
  - Tree structures (bidimensional lists)
- Example programming languages that provide pattern matching are Haskell, ML, Mathematica, Prolog

Pattern Matching of Types in C#

- C# **does not provide** pattern matching
- A naïve approach to provide pattern matching is by means of **introspection**: the `is` operator and/or the `GetType` method

```
static double AreaIs(Object figure) {
  if (figure is Circle)
    return Math.PI * Math.Pow(((Circle)figure).Radius, 2);
  if (figure is Square)
    return Math.Pow(((Square)figure).Side, 2);
  if (figure is Rectangle) {
    Rectangle rectangle = figure as Rectangle;
    return rectangle.Height * rectangle.Width;
  }
  if (figure is Triangle) {
    Triangle triangle = figure as Triangle;
    return triangle.Base * triangle.Height / 2;
    }
  throw new ArgumentException("The parameter is not a figure");
}
```

Never write this code!!!
It is totally unmaintainable

- pattern.matching/types

Pattern Matching of Types

- The use of introspection has two drawbacks
  1. Runtime performance is low
  2. Type errors are not detected by the compiler (they are detected at runtime)
- Question: In C#, an object-oriented programming language,

  which is the most appropriate technique to obtain the benefits of pattern matching for types (previous example)?

Pattern Matching in F#

- F# is a ML dialect implementation on .NET
- ML makes extensive use of pattern matching
- The following ML *fibonacci* function uses pattern matching (sequence patterns)

```
let rec fib n =
    match n with
    | 1 -> 1
    | 2 -> 1
    | x -> fib(x - 2) + fib(x - 1)
    ;;
```

- pattern.matching/fsharp

Pattern Matching of Types in F#

- F# also provides pattern matching with tree patterns

```
type Figure =
| Circle of double
| Rectangle of double * double
| Square of double
| Triangle of double * double
;;
let area figure =
    match figure with
    | Circle(radius) -> 3.141592 * radius * radius
    | Rectangle(width, height) -> width * height
    | Square(side) -> side * side
    | Triangle(theBase, height) -> theBase*height/2.0
    ;;
```

- pattern.matching/fsharp

Pattern Matching of Types in F#

- Question: Identify how to represent the next two code sections in object oriented languages (C# and Java)

```
type Figure =
| Circle of double
| Rectangle of double * double
| Square of double
| Triangle of double * double
;;
```

Section 1

```
let area figure =
   match figure with
   | Circle(radius) -> 3.141592 * radius * radius
   | Rectangle(width, height) -> width * height
   | Square(side) -> side * side
   | Triangle(theBase, heigth) -> theBase*height/2.0
   ;;
```

Section 2

Patterns with Guards

- In ML (F#), **conditions** (**guards**) can be specified in patterns
  - The **when** keyword and the _ wildcard character are used for this purpose

```
let regular figure =
    match figure with
    // _ is a wildcard character representing any value
    | Square (_) -> true
    // true when width == height
    | Rectangle(width, height) when width = height -> true
    // false for the rest of rectangles
    | Rectangle(_, _) -> false
    // true if it is isosceles
    | Triangle(theBase, height) -> height =
                theBase * System.Math.Sqrt(3.0) / 2.0
    | _ -> false // false for the rest of figures (the circle)
    ;;
```

- pattern.matching/fsharp

Pattern Matching of Lists in F#

- ML (F#) also provides pattern matching of lists
- The `::` **operator** is used for this purpose
- The *`head::tail`* pattern means that
  - *`head`* is the first element in the list
  - And *`tail`* is the remaining list after removing the first element

```
let list = ["hello"; "world"; "welcome"; "to"; "ML"]
let rec concatenate list =
  match list with
  | head :: tail -> head + " " + concatenate tail
  | [] -> ""
  ;;
```

  - A **declarative** style (not imperative) is observed
- pattern.matching/fsharp

Questions

- Answer the questions regarding the following ML (F#) code

Which higher-order function is this?

Main difference with our implementation in C#?

```
let rec higherOrder list f =
    match list with
    | head :: tail -> f head :: higherOrder tail f
    | [] -> []
    ;;
printfn "%A" (higherOrder [1; 2; 3; 4; 5] ((+) 1))
```

Language feature?

What is shown in the console?

Language feature?

- pattern.matching/fsharp

Mandatory Activity

- **Three entities** are involved in pattern matching
  1. The **element** the pattern is applied to
  2. The **pattern**
  3. The **action** to be performed when the pattern is matched against an element
- The two last entities can be represented with functions
- Let *T* be the type of the element, then
  - The **pattern** is represented with `Predicate<T>`
  - The **action** is represented with `Func<T,TResult>`
- Therefore, it is possible to implement a `PatternMatch` class that simulates this feature (pattern matching) receiving the three entities
- Analyze, modify, execute and comprehend:
  - pattern.matching/pattern.matching

Typical Higher-Order Functions

- Recall, a **higher-order function** is a function that
  - Either receives a function as a parameter
  - Or returns a function as a result
- There are many higher-order functions
- Typical ones are:
  - **`Filter`**: applies a predicate to all the elements in a collection, returning another collection containing those elements for which the predicate returns true
  - **`Map`**: applies a function to all the elements in a collection, returning a new collection with the results of each invocation
  - **`Reduce`** (**`Fold`**, **`Accumulate`**, **`Compress`** o **`Inject`**): A function is applied to all the elements in a collection (given one order), returning a single value

Higher-Order Functions in .NET

- The .NET Framework 4.0 includes many **generic higher order functions** in **`System.Linq`**
  - Are extension methods of **`IEnumerable`**
  - Receive functions as parameters, using **`Predicate`**, **`Func`** and **`Action`** types
- These are the given names for the typical higher-order functions mentioned in the previous slide
  - **`Filter`** ⇒ **`Where`**
  - **`Map`** ⇒ **`Select`**
  - **`Reduce`** ⇒ **`Aggregate`**
- higher.order/

Seminar 5

- Seminar 5
  - Closures and LINQ

List Comprehensions

- A **list comprehension** is a language feature for creating a list based on existing lists
- It follows the *set-builder* notation of set theory (used in mathematics and logic)
- For example

  $S = \{ 2x \mid x \in N, x < 10 \wedge x\%2=0 \}$

- Languages such as Haskell, OCaml, F# or Python support list comprehensions
- Python example:

```
S = [2*x for x in range(10) if x%2==0]
```

- Question: Does C# support them?

LINQ

- LINQ is quite similar to list comprehensions

```
public static IEnumerable<uint> NaturalNumber(uint max) {
  uint n = 0;
  while (n<max)
    yield return n++;
}

static IEnumerable<uint> WithoutSyntaxSugar() {
 return NaturalNumber(10)
              .Where(x => x % 2 == 0)
              .Select(x => 2 * x);
}
```

list.comprehensions/

LINQ Syntax Sugar

- LINQ offers a **syntax sugared** version to make it closer to existing list comprehensions approaches:

```
static IEnumerable<uint> WithoutSyntaxSugar() {
  return NaturalNumber(10)
              .Where(x => x % 2 == 0)
              .Select(x => 2 * x);
}
static IEnumerable<uint> WithSyntaxSugar() {
  return
    from x in NaturalNumber(10)
    where x % 2 == 0
    select 2 * x;
}
```

$\{ 2x \mid x \in N, x < 10 \wedge x\%2=0 \}$

list.comprehensions/

- However, there is not always a translation for every expression with the first syntax (less expressive)

Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

Foundations of Functional Programming

Francisco Ortín Soler

University of Oviedo