Logic and Functional Programming

Lecture 1: Introduction

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

Organization

Weekly lecture.

Topics:

- Introduction to Functional Programming (7 weeks):
  - theoretical aspects: lambda calculus, etc.,
  - practical programming in Racket and Haskell
- Introduction to logic programming (7 weeks):
  - logical foundations, computational model
  - practical programming in Prolog: programming techniques; selected examples; efficiency issues.

Grading:

- in-class quizzes, individual assignments: 20%
- two partial exams: 25% FP; 25% LP
- Written exam: 30%

References

Books

For Functional Programming:

- H. Abelson, G. J. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press Ltd. 1996.
- M. Marin, V. Negru, I. Drămnesc. *Principles and Practice of Functional Programming*. Editura UVT. 2016.
- S. Thompson. *Haskell: The Craft of Functional Programming*. Second Edition. Pearson Education Ltd. 1999.
- Haskell tutorials

For Logic Programming:

- W.F. Clocksin and C.S. Mellish. Programming in Prolog. Fifth edition. Springer. 2003.
- M.A. Covington *et al*. Coding Guidelines for Prolog. Theory and Practice of Logic Programming. 12(6): 889-927 (2012) (Highly recommended).
- P. Gloess. Constraint Logic Programming (PowerPoint format).

References

Software

For Functional Programming:

- Racket
- Haskell

For Logic Programming

- SWI-Prolog. For Windows users, there is a convenient SWI-Prolog editor.

Getting started

Imperative versus declarative

In the imperative style of programming

- the programmer should describe how to get the output for the range of required inputs.
- the programming process is based on the description of solutions to problems rather than on description of the problems themselves.

In the declarative style of programming

- the programmer should describe what relationships hold between various entities.
- the programming process is based on the description of the problems themselves.

Programming styles

There are two main imperative programming styles:

Procedural programming: programming with procedures=parameterized groups of instructions

Object-oriented programming: programming with objects = (usually) instances of classes.

This lecture is about the declarative programming styles:

Logic programming: programming using predicates.

Functional programming: programming using functions.

Functions versus procedures

In Functional Programming, functions are “pure”: their evaluation can not alter the environment of the computation

- $\Rightarrow$ no side effects.
- $\Rightarrow$ referentially transparent: the same function call always produces the same output.

In Procedural Programming, procedure calls can refer to global data, whose destructive assignment is also allowed.

- $\Rightarrow$ side effects
- $\Rightarrow$ referentially opaque: the same procedure call can produce different outputs.

Functional Programming

Computations and Programs

Main idea: Express every computation as a request to evaluate an expression, and use the resulting value for something.

- Program: collection of function definitions in the mathematical sense
  - The computation of a result depends only on the values of the function arguments, and not on the program state.
- Computation = evaluation of (nested) function calls.
- Each expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of a program to share the expression.
  - Evaluation of the expression changes its form, not value (e.g., 1+1 vs 2).
  - All references to the value are equivalent to the value itself.

Recursion

- Functions may be defined recursively, referring to themselves in their definition.
- In declarative programming, recursion is used for iteration.

Example (Computing the factorial)

Imperative style

```
int fact(int n) {
 int r=1,i=n;
 while(i>=1) {
   r=r*i;
   i=i-1;
 }
 return r;
}
```

Functional style

```
int fact(int n) {
  if(n==0)
     return 1;
   else
     return n*fact(n-1);
}
```

REMARK: iterative computations can be simulated by recursion.

Evaluation strategies

Every (functional) language implements a particular evaluation strategy. Function calls can be evaluated strictly (call-by-value) or lazily (call-by-name)

- Evaluation strategies affect efficiency and termination.

Example

Consider the function: $\mathtt{double}(\mathtt{x}) = \mathtt{x} + \mathtt{x}$.

- strict: $\mathtt{double}(1+2) = \mathtt{double}(3) = 3+3 = 6$.
- lazy:
  $\mathtt{double}(1+2) = (1+2)+(1+2) = 3+(1+2) = 3+3 = 6$.

First-class citizens and higher-order functions

- A first-class citizen is something that can be
  - named,
  - passed as argument to a function,
  - returned as result of a function call,
  - stored in a data structure (e.g., as element of a list)
- A higher-order function is a function that takes function(s) as argument(s) and/or returns a function as value.
- In functional programming:
  - Functions are first-class citizens
  - Functions can be higher-order

Mutable versus immutable data

Mutable data can be modified after its initial construction, immutable data can not be modified.

- Declarative programming uses immutable data.
- Imperative programming uses mutable data.

Declarative versus imperative programming styles

Summary of major differences

Declarative	Imperative
Focus on ”what”	Focus on “how”
Stateless	Uses state
Functions without side effects	Functions with side effects
Uses recursion to iterate	Uses loops to iterate
Good for Big Data	No good for Big Data
Statement execution order: not very important	Statement execution order: very important

Pros and cons of functional programming

Pros

- "No state, no side-effects" in functional programming help to write bugs-free code or less error-prone code.
- Functional code is compact, easier to maintain, reuse, and test.
- Functional programs consist of independent blocks that can run concurrently $\Rightarrow$ improved efficiency.
- They are close to mathematics, which is advantageous when proving their properties.

Recommended reading:

- J. Hughes. Why Functional Programming Matters. 1984.

Pros and cons of functional programming

Cons

The absence of state requires to create new objects whenever we perform actions

$\Rightarrow$ Functional Programming requires large memory space.

$\Rightarrow$ Garbage collection must be used to reclaim memory occupied by objects that become inaccessible.

- Historical note: Garbage collection was invented in 1959 by John McCarthy, the inventor of Lisp, the second-oldest high-level programming language (after Fortran) and the first functional programming language.

- Recursion is usually slower than iteration.
  - This is not so bad: modern languages have efficient garbage collectors $\Rightarrow$ some recursive computations can be as fast as iterative computations.

Languages

- Functional programming languages: Clean, Clojure, Haskell, Scala, Common Lisp, Scheme, Racket, ML family (including OCaml), Mathematica
- Languages that support functional programming style: Javascript, Lua, Oz, Python, ...

Functional versus logic programming

Functional Programming evaluates an expression according to a fixed, predictable strategy.

- expression = nested function calls
- program = collection of function definitions
- strategy: lazy or strict evaluation

Logic Programming answers a question by search according to a fixed, predictable strategy.

- question = conjunction of atomic queries.
- program = collection of predicate definitions
- strategy: SLDNF resolution

Logic Programming

Logic programming is based on a way of thinking which is useful for solving problems related to the extraction of knowledge from basic facts and relations:

- The programmer must describe what he knows as facts and rules collected in a program.
- The compiler (or interpreter) of the programming language finds the answers to all questions we may ask afterwards, using a built-in search strategy.

- Most representative language: Prolog
- search strategy of Prolog: SLDNF-resolution

Functional versus logic programming

Illustrated example

Compute/Find the minimum element of a list of numbers, using the following knowledge:

- Fact: The minimum element of a singleton list made of number m is m.
- Rules:
  - The minimum of x and y is x if $x \leq y$.
  - The minimum of x and y is y if $y < x$.
  - The minimum element of a list starting with x, y followed by sublist t is m if m is the minimum of x and n, where n is the minimum element of the list with first element y followed by sublist t.

Functional versus logic programming

Illustrated example

Functional Programming: Haskell

```
minList (x:[]) = x
minList (x:y)  = minim x (minList y)
minim x y
  | x <= y = x
  | x > y = y
```

Logic Programming: Prolog

```
min(X,Y,X) :- X =< Y.
min(X,Y,Y) :- Y < X.
minList([X],X).
minList([X|T],M) :- minList(T,Y), min(X,Y,M).
```

A word of warning

Functional Programming (FP) and Logic Programming (LP) are **declarative** programming styles:

- Programming = encode "what" you know in a program, without caring too much how the result/answer is computed
  - trust the built-in strategy of the language (FP: evaluation strategy; LP: resolution strategy). which always finds the right result/answer
- **Good thing:** Declarative programs are referentially transparent: they are easy to understand and verify if they are correct (with equational reasoning tools)
- **Bad thing:** Declarative programs can become very inefficient
  - We should care how the computation proceeds, and inmrove its efficiency by writing tail-recursive code.

Efficiency in FP

Example: computing Fibonacci numbers

Recursive: easy to understand but awfully inefficient implementation (in Haskell):

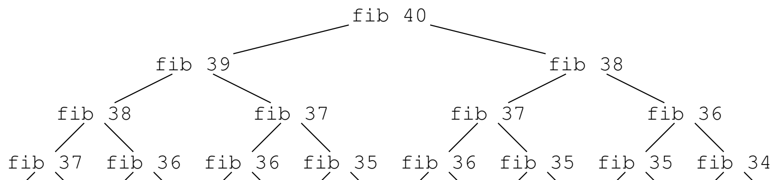
```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Tail recursive: less readable but very efficient

```
fib n = fib_acc n 1 1 where
  fib_acc 1 _    acc = acc
  fib_acc n prev acc = fib_acc (n-1) acc (prev+acc)
```

Example: computing Fibonacci numbers

Trace of computations



⇒ `fib 40` performs 331 160 281 recursive function calls!

`fib_acc 40 1 1` = `fib_acc 39 1 2` = `fib_acc 38 2 3` = ...
= `fib_acc 1 102334155 165580141` = `165580141`

⇒ `fib_acc 40 1 1` performs only 40 recursive function calls.

Functional programming languages

Lisp and its dialects

1955: John McCarthy (MIT): proposed the study of Artificial Intelligence (AI): "the science and engineering of making intelligent machines."

- Inventor of Lisp (1958) = first language with notable functional programming capabilities
  - Second oldest high-level programming language–only Fortran is 1 year older (from 1957)
  - Both Fortran and Lisp are in widespread use today
  - Lisp stands for **Lis**t **p**rocessing: linked lists are the main data structure, used to represent both source code (programs) and data.
  - Lists were used mainly for algebraic processing in AI
  - Other data types (besides lists): numbers and symbols
- Initially, Lisp was not standardized: many people developed their own versions of Lisp (a.k.a.Lisp dialects) $\Rightarrow$ standardization became necessary.

Functional programming languages

Major dialects of Lisp

There are 2 main dialects of Lisp, standardized and in widespread use:

1. Common Lisp: industrial standard developed by the Lisp community to combine the features from earlier Lisp dialects; became an ANSI standard in 1994
    - Huge, multi-paradigm programming language
2. Scheme: a Lisp dialect developed at MIT for instructional use; became an IEEE standard in 1990 (IEEE 1990), and was recently renamed to RACKET
    - Small, modular, easy-to-learn programming language
    - We will practice functional programming in RACKET (a dialect of Lisp) and Haskell

Peculiarities of Racket and Haskell

Racket, and all dialects of Lisp, use a weird syntax to write expressions, called fully parenthesised syntax. For example:

- Instead of $f(v_1, \ldots, v_n)$ we write $(f\ v_1\ \ldots\ v_n)$
- Instead of `if` *cond* `then` $branch_1$ `else` $branch_2$ we write `(if cond` $branch_1$ $branch_2$`)` etc.

Haskell requires the usage of parentheses only for two purposes: (1) to disambiguate the order of operator application (e.g., in arithmetic expressions), and (2) to build tuples (a composite datatype). For example:

- Instead of $f(v_1, \ldots, v_n)$ we write $f\ v_1\ \ldots\ v_n$

Evaluation strategies

Strict and lazy languages

Most functional languages (including Racket) are strict:

- Whenever we evaluate a function call, we first evaluate all function arguments to values, and then call the function with the values of the arguments:

  EXAMPLE:

  $$(+\ \ (/\ \ 4\ \ \underline{(-\ \ 3\ \ 1)})\ \ (*\ \ 2\ \ 5)) = (+\ \ \underline{(/\ \ 4\ \ 2)}\ \ (*\ \ 2\ \ 5)) = (+\ \ 2\ \ \underline{(*\ \ 2\ \ 5)}) = \underline{(+\ \ 2\ \ 10)} = 12$$

Sometimes, argument evaluation is useless:

$$(*\ \ 0\ \ (/\ \ (-\ \ (\mathrm{sqrt}\ \ (-17\ \ 1))\ \ (-\ \ 3\ \ 1)))) = 0$$

The evaluation of red argument is time-consuming and useless

- Lazy functional languages evaluate only needed arguments
  - Representative language: Haskell (standardized in 1990)

What is Racket?

A strict functional programming language: the entire language is built on top of a few primitive operations for list manipulation.

- enormous volume of educational material which created for it.
- Easy to get.

Easiest way to interact with Racket, is via DrRacket = widely used IDE among introductory Computer Science courses that teach Scheme or Racket

- Freely available for all major platforms: Windows, MacOS, UNIX, Linux with X Window system
- Recommended textbooks:
  - "Structure and Interpretation of Computer Programs", arguably the best textbook about functional programming.
  - "How to Design Programs" (from `http://www.htdp.org`)

More about strict functional programming

and Racket, in particular

- Every expression is evaluates to a value, by a stepwise process called reduction.
- Values are expressions that evaluate to themselves.
  - Can be primitive or composite
  - A **function expression** is evaluated to a function object
- Racket is dynamically typed:
  - We don't have to declare the types of variables, functions, etc.
  - The interpreter computes the types of expressions at runtime.
- Type = set of values with common properties.
  - type checking is performed at runtime, and can raise runtime type errors.

What is Haskell?

A lazy functional language created in the 1980's by a committee of academicians:

Functional Functions are first-class citizens: they are values which can be used as any other sort of value.

Lazy: Computation = evaluation of expressions using lazy evaluation

- expressions are not evaluated until their results are actually needed

Pure: Expressions are referentially transparent:

- no side effects
- calling the function with same inputs produces the same output every time

Statically typed: every expression has a type, which is checked at compile-time. Programs with type errors will not run because they will not even compile.

Main features of Haskell

1. Types

The type system is much more expressive in Haskell than in C++ or Java

- It helps clarify thinking and express program structure
  - Usually, the first step in writing a program is to write down all the types.
- Serves as a form of documentation
  - The type of a function tells you a lot about what a function might do and how it can be used.
- Turns run-time errors into compile-time errors
  - Many errors can be detected at compile time, and easy to fix. Run-time errors are hard to debug.

Main features of Haskell

2. Abstraction (“Don’t repeat yourself”))

- According to the Principle of Abstraction, nothing should be duplicated: every idea, algorithm, and piece of data should occur exactly once in your code.
- The process of abstraction: take similar pieces of code and factor out their commonality.

The following features of Haskell help us write abstract code (without repetitions):

- parametric polymorphism
- higher-order functions
- type classes

Main features of Haskell

3. Wholemeal programming

"Functional languages excel at wholemeal programming, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; develop a solution space, rather than an individual solution; imagine a graph, rather than a single path. The wholemeal approach often offers new insights or provides new perspectives on a given problem. It is nicely complemented by the idea of projective programming: first solve a more general problem, then extract the interesting bits and pieces by transforming the general program into more specialised ones."

Main features of Haskell

3. Wholemeal programming

Compare the following code in Java or C++:

```
int acc = 0;
for ( int i = 0; i < lst.length; i++ ) {
  acc = acc + 3 * lst[i];
}
```

with the equivalent Haskell code

```
sum (map (3*) lst)
```

- We will explore the shift in thinking represented by this way of programming, and how it works in functional programming.

Setting up Haskell

For Functional Programming, we will use GHCi, the interactive environment of the Glasgow Haskell Compiler.

- Freely available for all platforms. See Downloads for downloading instructions.
- GHCi Users Guide