The Meta Language (ML) and Functional Programming

Daniel S. Fava
danielsf@ifi.uio.no

Department of informatics
University of Oslo, Norway

Motivation ML Demo

Which programming languages are functional?

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)
- Lisp 1950

- ML 1973

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)
- Lisp 1950 dynamically typed

- ML 1973 statically typed

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)
- Lisp 1950 dynamically typed
  - Scheme 1970
    - Racket 1995
- ML 1973 statically typed

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)
- Lisp 1950 dynamically typed
  - Scheme 1970
    - Racket 1995
- ML 1973 statically typed
  - Standard ML or SML
  - Caml 1985
    - OCaml 1996

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)
- Lisp 1950 dynamically typed
  - Scheme 1970
    - Racket 1995
- ML 1973 statically typed
  - Standard ML or SML
  - Caml 1985
    - OCaml 1996
- Haskell 1990

Which programming languages are functional?

- Lambda calculus 1930 (a model of computation)
- Lisp 1950 dynamically typed
  - Scheme 1970
    - Racket 1995
- ML 1973 statically typed
  - Standard ML or SML
  - Caml 1985
    - OCaml 1996
- Haskell 1990
- Erlang 1986, Scala 2004, Clojure 2007 (multi-paradigm)

Why another programming paradigm?

Why another programming paradigm?

- A different way of thinking

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...
  - MapReduce

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...
  - MapReduce

Debatable

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...
  - MapReduce

Debatable

- easier to reason about and to prove properties of

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...
  - MapReduce

Debatable

- easier to reason about and to prove properties of
- more compact, simpler code

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...
  - MapReduce

Debatable

- easier to reason about and to prove properties of
- *in the eye of the beholder*
- more compact, simpler code

Why another programming paradigm?

- A different way of thinking
- Non-functional PLs borrowing functional constructs
  - JavaScript, Python, Lua, PHP, C++11, C# ...
  - MapReduce

Debatable

- easier to reason about and to prove properties of
- *in the eye of the beholder*
- more compact, simpler code
- problems can be easier to express given certain idioms
  match the language to the problem

What is functional programming?

What is functional programming?

It is programming with:

What is functional programming?

It is programming with:

- first-class and higher-order functions

What is functional programming?

It is programming with:

- first-class and higher-order functions
- purity

What is functional programming?

It is programming with:

- first-class and higher-order functions
- purity
- pattern matching

What is functional programming?

It is programming with:

- first-class and higher-order functions
- purity
- pattern matching
- recursion

- first-class and higher-order functions
- purity
- pattern matching
- recursion

- first-class and higher-order functions
  - Currying
  - Closures
- purity
- pattern matching
- recursion

- first-class and higher-order functions
  - Currying
  - Closures
- purity
  - Statements versus Expressions and Declarations
  - Side effects versus Effect free
  - Immutability
  - Referential transparency
- pattern matching

- recursion

- first-class and higher-order functions
  - Currying
  - Closures
- purity
  - Statements versus Expressions and Declarations
  - Side effects versus Effect free
  - Immutability
  - Referential transparency
- pattern matching
  - constructors and pattern matching
  - data structures
    - Nat, Lists, Trees, etc
- recursion

- first-class and higher-order functions
  - Currying
  - Closures
- purity
  - Statements versus Expressions and Declarations
  - Side effects versus Effect free
  - Immutability
  - Referential transparency
- pattern matching
  - constructors and pattern matching
  - data structures
    - Nat, Lists, Trees, etc
- recursion
  - tail recursion
  - termination and total functions
  - continuations and continuation passing style (CPS)

The Meta Language (ML)

ML is a functional programming language with imperative features,

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

Logic for Computable Functions (LCF)

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

Logic for **C**omputable **F**unctions (LCF)

Project headed by Robin Milner in 1972.

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

Logic for **C**omputable **F**unctions (LCF)

Project headed by Robin Milner in 1972.

- notation for programs

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

Logic for **C**omputable **F**unctions (LCF)

Project headed by Robin Milner in 1972.

- notation for programs
- notation for proofs (eg. proof that quicksort returns a sorted array)

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

Logic for **C**omputable **F**unctions (LCF)

Project headed by Robin Milner in 1972.

- notation for programs
- notation for proofs (eg. proof that quicksort returns a sorted array)
- write programs that search for proofs

The Meta Language (ML)

ML is a functional programming language with imperative features, it was part of project to create an *interactive theorem prover.*

Logic for **C**omputable **F**unctions (LCF)

Project headed by Robin Milner in 1972.

- notation for programs
- notation for proofs (eg. proof that quicksort returns a sorted array)
- write programs that search for proofs
- have a program that checks a proof

$$\texttt{tactic} : \texttt{formula} \to \texttt{proof}$$

$$\texttt{tactic} : \texttt{formula} \to \texttt{proof}$$

Composition of tactics

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:
  - terminate and return a proof, or

$$\texttt{tactic} : \texttt{formula} \rightarrow \texttt{proof}$$

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:
  - terminate and return a proof, or
  - diverge, which means run forever, or

`tactic : formula` $\rightarrow$ `proof`

Composition of tactics

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:
  - terminate and return a proof, or
  - diverge, which means run forever, or
  - raise an exception

`tactic : formula → proof`

Composition of tactics **Higher-order functions**

- Take tactics as input and return a more complex tactic

Proof checking

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:
  - terminate and return a proof, or
  - diverge, which means run forever, or
  - raise an exception

`tactic : formula → proof`

Composition of tactics — **Higher-order functions**

- Take tactics as input and return a more complex tactic

Proof checking — **Expressive and sound type system**

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:
  - terminate and return a proof, or
  - diverge, which means run forever, or
  - raise an exception

`tactic : formula → proof`

Composition of tactics — Higher-order functions

- Take tactics as input and return a more complex tactic

Proof checking — Expressive and sound type system

- `theorem` abstract data type
- theorems derivable only via inference rules given by `theorem`
- calling `tactic` on a `formula` can either:
  - terminate and return a proof, or
  - diverge, which means run forever, or
  - raise an exception — First language with type-safe exceptions

Read-Evaluate-Print Loop (REPL)

Read-Evaluate-Print Loop (REPL)

In ML, expressions are type-checked after **R** and before **E**