Concepts of Functional Programming

Neil Sculthorpe

Department of Computer Science
Royal Holloway, University of London
neil.sculthorpe@rhul.ac.uk

Nottingham, England
1st April 2016

What is Functional Programming?

- Functional programming is a paradigm.
- Based on evaluating expressions, not executing commands.
- Functional programs express output as a function of input, rather than as a sequence of steps to be performed.

Why Functional Programming?

- Concise programs
- Code reuse
- Types provide more compile-time checks
- Fewer bugs
- Rapid prototyping

Overview

- Higher-order Functions
- Purity
- Recursion
- Algebraic Data Types
- Lazy Evaluation
- Polymorphism
- Dependent Types

Function Syntax (Unary)

C-like languages

```
int sqr(int i) {
  return i * i;
}
```

Mathematics

$$sqr : \mathbb{Z} \to \mathbb{Z}$$
$$sqr(i) = i \times i$$

Functional language (Haskell)

$$sqr :: Int \to Int$$
$$sqr\ i = i * i$$

Function Syntax (Binary)

C-like languages

```
int hypot(int i, int j) {
    return sqrt(sqr(i)+sqr(j));
}
```

Functional language (Haskell)

$$hypot :: Int \rightarrow Int \rightarrow Int$$
$$hypot\ i\ j = sqrt\ (sqr\ i + sqr\ j)$$

Higher-order Functions

- Functions are first-class values.
- Higher-order functions take functions as arguments and/or return functions as results.

Higher-order Functions

- Functions are first-class values.
- Higher-order functions take functions as arguments and/or return functions as results.

$$apply :: (Int \rightarrow Int) \rightarrow Int \rightarrow Int$$
$$apply\ f\ i = f\ i$$

Higher-order Functions

- Functions are first-class values.
- Higher-order functions take functions as arguments and/or return functions as results.

$$\mathit{apply} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int}$$
$$\mathit{apply}\ f\ i = f\ i$$

$$\mathit{second} :: (\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \to \mathit{Int})$$
$$\mathit{second}\ f\ g = g$$

Higher-order Functions - Lambda Abstractions

- Lambda abstractions construct anonymous functions inline.
- Syntax: $(\lambda\ \textsc{Argument} \rightarrow \textsc{Result})$

Higher-order Functions - Lambda Abstractions

- Lambda abstractions construct anonymous functions inline.
- Syntax: $(\lambda\ \textsc{Argument} \rightarrow \textsc{Result})$
- E.g.

$$
\begin{aligned}
&x :: Int \\
&x = apply\ sqr\ 3 \\
&sqr :: Int \rightarrow Int \\
&sqr\ i = i * i
\end{aligned}
$$

is equivalent to

$$
\begin{aligned}
&x :: Int \\
&x = apply\ (\lambda i \rightarrow i * i)\ 3
\end{aligned}
$$

Higher-order Functions - Manipulating Functions

$$compose :: (A \to B) \to (B \to C) \to (A \to C)$$
$$compose\ f\ g = \lambda a \to g\ (f\ a)$$

$$flip :: (A \to B \to C) \to (B \to A \to C)$$
$$flip\ f = \lambda b\ a \to f\ a\ b$$

Purity

- The result of a pure function is completely determined by its arguments.

Purity

- The result of a pure function is completely determined by its arguments.
- A pure function thus has no computational side effects.

Purity

- The result of a pure function is completely determined by its arguments.
- A pure function thus has no computational side effects.
- Advantages for: clarity, testing, debugging, refactoring, optimisation, trustworthiness.

Purity

- The result of a pure function is completely determined by its arguments.
- A pure function thus has no computational side effects.
- Advantages for: clarity, testing, debugging, refactoring, optimisation, trustworthiness.
- Data structures are immutable.

Purity

- The result of a pure function is completely determined by its arguments.
- A pure function thus has no computational side effects.
- Advantages for: clarity, testing, debugging, refactoring, optimisation, trustworthiness.
- Data structures are immutable.
- No imperative variables!

Recursion

- Functional programs use recursion instead of iteration.

Recursion

- Functional programs use recursion instead of iteration.

C-like language

```
int fact(int i) {
   int result = 1;
   for (int n = i; n > 1; n--) {
     result = result * n;
   }
   return result;
}
```

Recursion

- Functional programs use recursion instead of iteration.

C-like language

```
int fact(int i) {
   int result = 1;
   for (int n = i; n > 1; n--) {
     result = result * n;
   }
   return result;
}
```

Functional language (Haskell)

$$
\begin{aligned}
&\mathit{fact} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fact}\ 0 = 1 \\
&\mathit{fact}\ i\ = i * \mathit{fact}\ (i - 1)
\end{aligned}
$$

Algebraic Data Types

- Functional data structures are built using algebraic data types.

Algebraic Data Types

- Functional data structures are built using algebraic data types.
- Formed from sums, products and recursion.

Algebraic Data Types

- Functional data structures are built using algebraic data types.
- Formed from sums, products and recursion.
- Functions can be defined by pattern matching.

Algebraic Data Types

- Functional data structures are built using algebraic data types.
- Formed from sums, products and recursion.
- Functions can be defined by pattern matching.
- Days of the week:

$$\mathbf{data}\ \mathit{Day} = \mathrm{Mon} \mid \mathrm{Tue} \mid \mathrm{Wed} \mid \mathrm{Thu} \mid \mathrm{Fri} \mid \mathrm{Sat} \mid \mathrm{Sun}$$

Algebraic Data Types

- Functional data structures are built using algebraic data types.
- Formed from sums, products and recursion.
- Functions can be defined by pattern matching.
- Days of the week:

$$\textbf{data}\ \mathit{Day} = \mathrm{Mon} \mid \mathrm{Tue} \mid \mathrm{Wed} \mid \mathrm{Thu} \mid \mathrm{Fri} \mid \mathrm{Sat} \mid \mathrm{Sun}$$

$$\begin{aligned}
&\mathit{isWeekend} :: \mathit{Day} \to \mathit{Bool}\\
&\mathit{isWeekend}\ \mathrm{Mon} = \mathrm{False}\\
&\mathit{isWeekend}\ \mathrm{Tue} = \mathrm{False}\\
&\mathit{isWeekend}\ \mathrm{Wed} = \mathrm{False}\\
&\mathit{isWeekend}\ \mathrm{Thu} = \mathrm{False}\\
&\mathit{isWeekend}\ \mathrm{Fri} = \mathrm{False}\\
&\mathit{isWeekend}\ \mathrm{Sat} = \mathrm{True}\\
&\mathit{isWeekend}\ \mathrm{Sun} = \mathrm{True}
\end{aligned}$$

Algebraic Data Types

- Functional data structures are built using algebraic data types.
- Formed from sums, products and recursion.
- Functions can be defined by pattern matching.
- Days of the week:

$$\textbf{data}\ \mathit{Day} = \mathrm{Mon} \mid \mathrm{Tue} \mid \mathrm{Wed} \mid \mathrm{Thu} \mid \mathrm{Fri} \mid \mathrm{Sat} \mid \mathrm{Sun}$$

$$\begin{array}{lll}
\mathit{isWeekend} :: \mathit{Day} \to \mathit{Bool} & & \\
\mathit{isWeekend}\ \mathrm{Sat} & = & \mathrm{True} \\
\mathit{isWeekend}\ \mathrm{Sun} & = & \mathrm{True} \\
\mathit{isWeekend}\ _ & = & \mathrm{False}
\end{array}$$

Algebraic Data Types

- Functional data structures are built using algebraic data types.
- Formed from sums, products and recursion.
- Functions can be defined by pattern matching.
- Days of the week:

$$\textbf{data}\ \mathit{Day} = \mathrm{Mon} \mid \mathrm{Tue} \mid \mathrm{Wed} \mid \mathrm{Thu} \mid \mathrm{Fri} \mid \mathrm{Sat} \mid \mathrm{Sun}$$

$$\begin{array}{lll} \mathit{isWeekend} :: \mathit{Day} \to \mathit{Bool} & & \\ \mathit{isWeekend}\ \mathrm{Sat} & = & \mathrm{True} \\ \mathit{isWeekend}\ \mathrm{Sun} & = & \mathrm{True} \\ \mathit{isWeekend}\ \_ & = & \mathrm{False} \end{array}$$

- Binary tree with integer leaves:

$$\textbf{data}\ \mathit{BinaryTree} = \mathrm{Leaf}\ \mathit{Int} \mid \mathrm{Node}\ \mathit{BinaryTree}\ \mathit{BinaryTree}$$

Algebraic Data Types - Singly Linked Lists

$\mathbf{data}\ \mathit{List} = \mathrm{Nil} \mid \mathrm{Cons}\ \mathit{Int}\ \mathit{List}$

Algebraic Data Types - Singly Linked Lists

$$\textbf{data}\ \mathit{List} = \mathrm{Nil} \mid \mathrm{Cons}\ \mathit{Int}\ \mathit{List}$$

$$\begin{aligned}
&\mathit{sqrList} :: \mathit{List} \to \mathit{List} \\
&\mathit{sqrList}\ \mathrm{Nil} &&= \mathrm{Nil} \\
&\mathit{sqrList}\ (\mathrm{Cons}\ i\ l) &&= \mathrm{Cons}\ (\mathit{sqr}\ i)\ (\mathit{sqrList}\ l)
\end{aligned}$$

Algebraic Data Types - Singly Linked Lists

$$\textbf{data}\ \mathit{List} = \mathrm{Nil} \mid \mathrm{Cons}\ \mathit{Int}\ \mathit{List}$$

$$\begin{aligned}
&\mathit{sqrList} :: \mathit{List} \to \mathit{List} \\
&\mathit{sqrList}\ \mathrm{Nil} = \mathrm{Nil} \\
&\mathit{sqrList}\ (\mathrm{Cons}\ i\ l) = \mathrm{Cons}\ (\mathit{sqr}\ i)\ (\mathit{sqrList}\ l)
\end{aligned}$$

$$\begin{aligned}
&\mathit{mapList} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{List} \to \mathit{List} \\
&\mathit{mapList}\ f\ \mathrm{Nil} = \mathrm{Nil} \\
&\mathit{mapList}\ f\ (\mathrm{Cons}\ i\ l) = \mathrm{Cons}\ (f\ i)\ (\mathit{mapList}\ f\ l)
\end{aligned}$$

Algebraic Data Types - Singly Linked Lists

$$\textbf{data}\ \mathit{List} = \mathrm{Nil} \mid \mathrm{Cons}\ \mathit{Int}\ \mathit{List}$$

$$\begin{array}{l} \mathit{sqrList} :: \mathit{List} \to \mathit{List} \\ \mathit{sqrList}\ \mathrm{Nil} \qquad\qquad = \mathrm{Nil} \\ \mathit{sqrList}\ (\mathrm{Cons}\ i\ l) = \mathrm{Cons}\ (\mathit{sqr}\ i)\ (\mathit{sqrList}\ l) \end{array}$$

$$\begin{array}{l} \mathit{mapList} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{List} \to \mathit{List} \\ \mathit{mapList}\ f\ \mathrm{Nil} = \mathrm{Nil} \\ \mathit{mapList}\ f\ (\mathrm{Cons}\ i\ l) = \mathrm{Cons}\ (f\ i)\ (\mathit{mapList}\ f\ l) \end{array}$$

$$\begin{array}{l} \mathit{sqrList} :: \mathit{List} \to \mathit{List} \\ \mathit{sqrList}\ l = \mathit{mapList}\ \mathit{sqr}\ l \end{array}$$

Algebraic Data Types - Singly Linked Lists

$\textbf{data}\ \mathit{List} = \mathrm{Nil} \mid \mathrm{Cons}\ \mathit{Int}\ \mathit{List}$

$\mathit{sqrList} :: \mathit{List} \rightarrow \mathit{List}$
$\mathit{sqrList}\ \mathrm{Nil} = \mathrm{Nil}$
$\mathit{sqrList}\ (\mathrm{Cons}\ i\ l) = \mathrm{Cons}\ (\mathit{sqr}\ i)\ (\mathit{sqrList}\ l)$

$\mathit{mapList} :: (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow \mathit{List} \rightarrow \mathit{List}$
$\mathit{mapList}\ f\ \mathrm{Nil} = \mathrm{Nil}$
$\mathit{mapList}\ f\ (\mathrm{Cons}\ i\ l) = \mathrm{Cons}\ (f\ i)\ (\mathit{mapList}\ f\ l)$

$\mathit{sqrList} :: \mathit{List} \rightarrow \mathit{List}$
$\mathit{sqrList}\ l = \mathit{mapList}\ \mathit{sqr}\ l$

$\mathit{sqrtList} :: \mathit{List} \rightarrow \mathit{List}$
$\mathit{sqrtList}\ l = \mathit{mapList}\ \mathit{sqrt}\ l$

Lazy Evaluation

- Purity enables lazy evaluation.

Lazy Evaluation

- Purity enables lazy evaluation.
- Defer evaluation of expressions until the result is needed.

Lazy Evaluation

- Purity enables lazy evaluation.
- Defer evaluation of expressions until the result is needed.
- Infinite data structures can be expressed directly.

Lazy Evaluation

- Purity enables lazy evaluation.
- Defer evaluation of expressions until the result is needed.
- Infinite data structures can be expressed directly.
- E.g.

$$\mathit{intsFrom} :: \mathit{Int} \rightarrow \mathit{List}$$
$$\mathit{intsFrom}\ i = \mathrm{Cons}\ i\ (\mathit{intsFrom}\ (i + 1))$$

Polymorphism (Parametric)

- Functional languages can infer the types of functions.

Polymorphism (Parametric)

- Functional languages can infer the types of functions.
- Functions can have polymorphic types.

$$apply :: (a \to b) \to a \to b$$
$$apply\ f\ x = f\ x$$

Polymorphism (Parametric)

- Functional languages can infer the types of functions.
- Functions can have polymorphic types.

$$
\begin{aligned}
&\mathit{apply} :: (a \to b) \to a \to b \\
&\mathit{apply}\ f\ x = f\ x
\end{aligned}
$$

- Data types can also be polymorphic:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{List}\ a = \mathrm{Nil} \mid \mathrm{Cons}\ a\ (\mathit{List}\ a) \\
&\mathit{mapList} :: (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b \\
&\mathit{mapList}\ f\ \mathrm{Nil} = \mathrm{Nil} \\
&\mathit{mapList}\ f\ (\mathrm{Cons}\ x\ l) = \mathrm{Cons}\ (f\ x)\ (\mathit{mapList}\ f\ l)
\end{aligned}
$$

Polymorphism (Ad-hoc)

- Ad-hoc polymorphic functions are restricted to classes of types.

$$sqr :: Num\ a \Rightarrow a \rightarrow a$$
$$sqr\ x = x * x$$

$$sort :: Ord\ a \Rightarrow List\ a \rightarrow List\ a$$
$$sort\ l = \ldots$$

Polymorphism (Ad-hoc)

- Ad-hoc polymorphic functions are restricted to classes of types.

$$sqr :: Num\ a \Rightarrow a \rightarrow a$$
$$sqr\ x = x * x$$
$$sort :: Ord\ a \Rightarrow List\ a \rightarrow List\ a$$
$$sort\ l = \ldots$$

- Type constructors can also be polymorphic.

Polymorphism (Ad-hoc)

- Ad-hoc polymorphic functions are restricted to classes of types.

$$sqr :: Num\ a \Rightarrow a \rightarrow a$$
$$sqr\ x = x * x$$
$$sort :: Ord\ a \Rightarrow List\ a \rightarrow List\ a$$
$$sort\ l = \ldots$$

- Type constructors can also be polymorphic.
  E.g. rather than,

$$\begin{aligned} &mapList &&:: (a \rightarrow b) \rightarrow List\ a &&\rightarrow List\ b \\ &mapTree &&:: (a \rightarrow b) \rightarrow Tree\ a &&\rightarrow Tree\ b \\ &mapVector &&:: (a \rightarrow b) \rightarrow Vector\ a &&\rightarrow Vector\ b \end{aligned}$$

Polymorphism (Ad-hoc)

- Ad-hoc polymorphic functions are restricted to classes of types.

$$sqr :: Num\ a \Rightarrow a \to a$$
$$sqr\ x = x * x$$
$$sort :: Ord\ a \Rightarrow List\ a \to List\ a$$
$$sort\ l = \ldots$$

- Type constructors can also be polymorphic.
  E.g. rather than,

$$\begin{aligned} mapList &:: (a \to b) \to List\ a &&\to List\ b \\ mapTree &:: (a \to b) \to Tree\ a &&\to Tree\ b \\ mapVector &:: (a \to b) \to Vector\ a &&\to Vector\ b \end{aligned}$$

  we can define a single general purpose *map*:

$$map :: Functor\ t \Rightarrow (a \to b) \to t\ a \to t\ b$$

Dependent Types

- Def: A function has a dependent type when the type of its result depends on the value of its argument.

Dependent Types

- Def: A function has a dependent type when the type of its result depends on the value of its argument.
- In practice: types and values can be intermixed.

Dependent Types

- Def: A function has a dependent type when the type of its result depends on the value of its argument.
- In practice: types and values can be intermixed.
- Extremely powerful for encoding compile-time checks in a program.

Dependent Types

- Def: A function has a dependent type when the type of its result depends on the value of its argument.
- In practice: types and values can be intermixed.
- Extremely powerful for encoding compile-time checks in a program.
- E.g. we could add the length of a list to its type:

$$map :: (a \to b) \to List\ n\ a \to List\ n\ b$$
$$append :: List\ m\ a \to List\ n\ a \to List\ (m + n)\ a$$
$$sort :: List\ n\ a \to List\ n\ a$$

Dependent Types

- Def: A function has a dependent type when the type of its result depends on the value of its argument.
- In practice: types and values can be intermixed.
- Extremely powerful for encoding compile-time checks in a program.
- E.g. we could add the length of a list to its type:

$$map :: (a \to b) \to List\ n\ a \to List\ n\ b$$
$$append :: List\ m\ a \to List\ n\ a \to List\ (m + n)\ a$$
$$sort :: List\ n\ a \to List\ n\ a$$

- Logical properties of the program can be encoded within the program itself.

Conclusion

- Functional programming involves several (complementary) concepts, including:
  - Higher-order Functions
  - Purity
  - Recursion
  - Algebraic Data Types
  - Lazy Evaluation
  - Polymorphism
  - Dependent Types
- Not all concepts appear in every functional language.
- Increasingly more of these concepts are being added to non-functional languages.