ST0244 Programming Languages
5. Functional Programming

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2023-2

Preliminaries

Conventions

- The number and page numbers assigned to chapters, examples, exercises, figures, quotes, sections and theorems on these slides correspond to the numbers assigned in the textbook [Lee 2017].
- The source code examples are in course's repository.

Introduction

Feature	Imperative	Functional
Assignment of variables	Yes	No
Iteration	Yes	No
Recursion	Possible	Necessary
Higher-order functions	Possible	Yes
First-class functions	No	Yes
Side-effects	Yes	Avoid or isolate
Theoretical model	Turing machine	Lambda calculus
Program execution	Execution of statements	Evaluation of expressions

Introduction

Description

> *‘A* ***side effect*** *introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.’ (O’Sullivan, Goerzen and Stewart 2008, p. 27)*

Introduction

Description

> *‘A **side effect** introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.’ (O’Sullivan, Goerzen and Stewart 2008, p. 27)*

Description

A **pure function** is a side-effect free function (e.g. does not cause mutation of mutable objects nor output to I/O devices). That is, pure functions

> *‘take all their input as explicit arguments, and produce all their output as explicit results.’ (Hutton 2016, § 10.1)*

Introduction

Description

> *‘A **side effect** introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.’ (O’Sullivan, Goerzen and Stewart 2008, p. 27)*

Description

A **pure function** is a side-effect free function (e.g. does not cause mutation of mutable objects nor output to I/O devices). That is, pure functions

> *‘take all their input as explicit arguments, and produce all their output as explicit results.’ (Hutton 2016, § 10.1)*

Example (C++ and Pascal)

See files `fp/side-effect*`.

Lambda Calculus

Introduction

- A formal system invented by Alonzo Church around 1930s.
- The goal was to use the λ-calculus in the foundation of mathematics.
- Intended for studying functions and recursion.
- Computability model.
- A free-type functional programming language.
- λ-notation (e.g. anonymous functions and currying).

Lambda Calculus

Application

Application of the function M to argument N is denoted by MN (juxtaposition).

Lambda Calculus

Application

Application of the function M to argument N is denoted by MN (juxtaposition).

Abstraction

> *'If M is any formula containing the variable x, then $\lambda x[M]$ is a symbol for the function whose values are those given by the formula.' (Church 1932, p. 352)*

Lambda Calculus

Currying

> *'Adopting a device due to Schönfinkel, we treat a function of two variables as a function of one variable whose values are functions of one variable, and a function of three or more variables similarly.' (Church 1932, p. 352)*

Such device is called **currying** after Haskell Curry.

(continued on next slide)

Lambda Calculus

Currying (continuation)

Let $g : X \times Y \to Z$ be a function of two variables. We can define two functions f_x and f:

$$f_x : Y \to Z \qquad\qquad f : X \to (Y \to Z)$$

$$f_x = \lambda y.g(x, y), \qquad\qquad f = \lambda x.f_x.$$

Then $(f\,x)\,y = f_x\,y = g(x, y)$. That is, the function of two variables

$$g : X \times Y \to Z$$

is represented as the higher-order function

$$f : X \to (Y \to Z).$$

Lambda Calculus

Definition

The set of **λ-terms** is described by

$$
\begin{aligned}
M, N ::= {} & x && \text{(variable)} \\
\mid {} & (\lambda x.M) && \text{(λ-abstraction)} \\
\mid {} & (MN) && \text{(application)}
\end{aligned}
$$

Lambda Calculus

Definition

The set of **λ-terms** is described by

$$\begin{aligned} M, N ::= {} & x && \text{(variable)} \\ & \mid (\lambda x.M) && \text{(λ-abstraction)} \\ & \mid (M N) && \text{(application)} \end{aligned}$$

Conventions

- λ-term variables will be denoted by $x, y, z, \ldots$.
- λ-terms will be denoted by $M, N, \ldots$.

Lambda Calculus

Definition

The set of **λ-terms** is described by

$$\begin{aligned} M, N ::=\ & x && \text{(variable)} \\ \mid\ & (\lambda x.M) && \text{(λ-abstraction)} \\ \mid\ & (M N) && \text{(application)} \end{aligned}$$

Conventions

- λ-term variables will be denoted by $x, y, z, \ldots$.
- λ-terms will be denoted by $M, N, \ldots$.

Example

Whiteboard.

Lambda Calculus

Conventions and syntactic sugar

- Outermost parentheses are not written.
- Application has higher precedence, that is,

$$\lambda x.MN := (\lambda x.(MN)).$$

- Application associates to the left, that is,

$$MN_1N_2\dots N_k := (\dots((MN_1)N_2)\dots N_k).$$

- Lambda abstraction associates to the right, that is,

$$\begin{aligned}\lambda x_1x_2\dots x_n.M &:= \lambda x_1.\lambda x_2.\dots\lambda x_n.M\\ &:= (\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots))).\end{aligned}$$

Lambda Calculus

Definition

The functional behaviour of the λ-calculus is formalised through of their reduction/conversion rules. The **β-reduction rule** is defined by

$$(\lambda x.M)N \Rightarrow M[\,x \mapsto N\,],$$

where $M[\,x \mapsto N\,]$ denotes the result of **substituting $\boldsymbol{N}$ for every free occurrence of $\boldsymbol{x}$ in $\boldsymbol{M}$**.[*]

Example

Whiteboard.

---

[*] See, e.g. [Barendregt 2004; Hindley and Seldin 2008].

Lambda Calculus

Definition

A **redex** is a λ-term of the form $(\lambda x.M)N$.

Definition

A λ-term which contains no redex is in **normal form**.

Lambda Calculus

Definition

A redex is an **outermost redex** iff it is not contained in any other redex.

A redex is an **innermost redex** iff it contains no other redex.

Example

Let $M := (\lambda y.z)((\lambda x.xx)(\lambda x.xx))$. Then

- M is an outermost redex.
- M is not an innermost redex because it contains a redex.
- $(\lambda x.xx)(\lambda x.xx)$ is an innermost redex.
- $(\lambda x.xx)(\lambda x.xx)$ is not an outermost redex because it is contained in a redex.

Lambda Calculus

Definition

The **normal order reduction** is the evaluation strategy where the left-most outermost redex is reduced first.

Lambda Calculus

Definition

The **normal order reduction** is the evaluation strategy where the left-most outermost redex is reduced first.

Definition

The **applicative order reduction** is the evaluation strategy where the left-most innermost redex is reduced first.

Lambda Calculus

Example

To reduce $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda xy.x)$ using both normal order reduction and applicative order reduction.

Normal order reduction

$$\begin{aligned}
&\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)\\
&\Rightarrow \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda xy.x)}\\
&\Rightarrow \lambda z.\underline{(\lambda x.x)z}((\lambda xy.x)z)\\
&\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})\\
&\Rightarrow \lambda z.z(\lambda y.z)
\end{aligned}$$

Applicative order reduction

$$\begin{aligned}
&\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)\\
&\Rightarrow (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda xy.x)\\
&\Rightarrow \underline{(\lambda yz.z(yz))(\lambda xy.x)}\\
&\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})\\
&\Rightarrow \lambda z.z(\lambda y.z)
\end{aligned}$$

Lambda Calculus

Example

Let $\Omega := (\lambda x.xx)(\lambda x.xx)$. To reduce $(\lambda y.z)\Omega$ using both normal order reduction and applicative order reduction.

Normal order reduction

$$
\begin{aligned}
&\underline{(\lambda y.z)\Omega}\\
&\Rightarrow z
\end{aligned}
$$

Applicative order reduction

$$
\begin{aligned}
&(\lambda y.z)\Omega\\
&= (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})\\
&\Rightarrow (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})\\
&\Rightarrow (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})\\
&\Rightarrow \ldots
\end{aligned}
$$

Lambda Calculus

Remark

Church [1935, 1936] proved that the set

$$\{\, M \in \text{λ-terms} \mid M \text{ has a normal form} \,\}$$

is undecidable. This was the first undecidable set ever.

Getting Started with Haskell

Introduction

Haskell is a functional language based on various functional languages which in turn are based on the λ-calculus. For a very complete history of this language see [Hudak, John Hughes, Peyton Jones and Wadler 2007].

Getting Started with Haskell

Important Haskell features*

- Haskell is a pure and lazy functional programming language.
- Haskell is higher-order supporting functions as first-class values.
- It is strongly typed like Pascal, but more powerful since it supports polymorphic type checking.

  Remark on the sentence:

  'With this strong type checking it is pretty infrequent that you need to debug your code!! What a great thing!!!' (p. 184)

(continued on next slide)

---

*Almost copy-paste from Section "5.3 Getting Started with Standard ML" in the textbook.

Getting Started with Haskell

Important Haskell features (continuation)

- It provides a safe environment for code development and execution. This means there are no traditional pointers in Haskell.
- Since there are no traditional pointers, garbage collection is implemented in the Haskell system.
- Pattern-matching is provided for conveniently writing recursive functions.
- Lists are a built-in data type.
- A library of commonly used functions and data structures is available called the Base Library.

Getting Started with Haskell

Suggested reading

J. Hughes [1989, p. 107] wrote:

> *'In this paper, we have argued that modularity is the key to successful programming [...] Functional programming languages provide two new kinds of glue—higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways [...] Smaller and more general modules can be re-used more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones.'*

Getting Started with Haskell

Suggested reading

J. Hughes [1989, p. 107] wrote:

> *'In this paper, we have argued that modularity is the key to successful programming [...] Functional programming languages provide two new kinds of glue—higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways [...] Smaller and more general modules can be re-used more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones.'*

Remark

The above paper was written in 1984 and it circulated as a memo. The paper did not use Haskell but Miranda, a predecessor of Haskell.

Expressions, Types, Functions and Guards

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Expressions, Types, Functions and Guards

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Question

Is the `factorial` function correct?

Expressions, Types, Functions and Guards

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Question

Is the `factorial` function correct?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Expressions, Types, Functions and Guards

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Question

Is the `factorial` function correct?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

From the type of the function we know the function is buggy. Why?

Expressions, Types, Functions and Guards

Example

One solution for the buggy `factorial` function using **guards**.

```
factorial :: Int -> Int
factorial n
  | n == 0    = 1
  | n >  0    = n * factorial (n - 1)
  | otherwise = error "factorial: n < 0"
```

Expressions, Types, Functions and Guards

Example

One solution for the buggy `factorial` function using **guards**.

```
factorial :: Int -> Int
factorial n
  | n == 0    = 1
  | n >  0    = n * factorial (n - 1)
  | otherwise = error "factorial: n < 0"
```

Other solutions (humor)

Google for ‘The evolution of a Haskell programmer’.

Currying

Functions for currying and uncurrying

(i) Converts an uncurried function to a curried function.

```
curry :: ((a, b) -> c) -> a -> b -> c
```

(ii) Converts a curried function to a function on pairs.

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Lists

Inductive definition

Haskell has built-in syntax for lists, where a list is either:

- the empty list, written `[]`, or
- a first element `x` and a list `xs`, written `(x : xs)`.

The operator ‘`:`‘ is usually called **cons**.

Lists

Example (recursive function using pattern matching on lists)

Returns the length of a finite list of `Int`'s as an `Int`.

```
lengthInt :: [Int] -> Int
lengthInt []       = 0
lengthInt (x : xs) = 1 + lengthInt xs
```

Lists

Example (recursive function using pattern matching on lists)

Returns the length of a finite list of `Int`'s as an `Int`.

```
lengthInt :: [Int] -> Int
lengthInt []       = 0
lengthInt (x : xs) = 1 + lengthInt xs
```

Question

What about the `length` function on lists of Booleans?

Lists

Example (recursive function using pattern matching on lists)

Returns the length of a finite list of `Int`'s as an `Int`.

```
lengthInt :: [Int] -> Int
lengthInt []       = 0
lengthInt (x : xs) = 1 + lengthInt xs
```

Question

What about the `length` function on lists of Booleans?

Returns the length of a finite list of `Bools`'s as an `Int`.

```
lengthBool :: [Bool] -> Int
lengthBool []       = 0
lengthBool (x : xs) = 1 + lengthBool xs
```

Lists

Question

Can we avoid the boilerplate code? Yes!

Parametric Polymorphism

Lists

The built-in lists are parametric polymorphics.

```
GHCi> :t []
[] :: [a]

GHCi> :t (:)
(:) :: a -> [a] -> [a]
```

Parametric Polymorphism

Example

Returns the length of a finite list (of any type) as an `Int`.

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Parametric Polymorphism

Example

Appends two lists.

```
(++) :: [a] -> [a] -> [a]
(++) []       ys = ys
(++) (x : xs) ys = x : xs ++ ys
```

Parametric Polymorphism

Example (functions from the basic library)

- Extracts the first element of a list, which must be non-empty.

```
head :: [a] -> a
```

- Extracts the last element of a list, which must be finite and non-empty.

```
last :: [a] -> a
```

- Extracts the elements after the head of a list, which must be non-empty.

```
tail :: [a] -> [a]
```

Parametric Polymorphism

Example (functions from the basic library)

- Returns all the elements of a list except the last one. The list must be non-empty.

```
init :: [a] -> [a]
```

- Tests whether a list is empty.

```
null :: [a] -> Bool
```

Recursion

Definition

A function is **recursive** iff it calls itself.

Recursion

Writing recursive functions (p. 188)

1. '*Decide what the function is named, what arguments are passed to it, and what the function should return.*'
2. '*At least one of the arguments must get smaller each time. Most of the time it is only one argument getting smaller. Decide which one that will be.*'
3. '*Write the function declaration, declaring the name, arguments types, and return type if necessary.*'
4. '*Write a base case for the argument that you decided will get smaller. Pick the smallest, simplest value that could be passed to the function and just return the result for that base case.*'
5. '*The next step is the crucial step. You don't write the next statement from left to right. You write from the inside out at this point.*'

(continued on next slide)

Recursion

Writing recursive functions (p. 188) (continuation)

6. ‘*Make a recursive call to the function with a smaller value. For instance, if it is a list you decided will get smaller, call the function with the tail of the list. If an integer is the argument getting smaller, call the function with the integer argument minus 1. Call the function with the required arguments and in particular with a smaller value for the argument you decided would get smaller at each step.*’
7. ‘*Now, here’s a leap of faith. That call you made in the last step worked! It returned the result that you expected for the arguments it was given. Use that result in building the result for the original arguments passed to the function. At this step it may be helpful to try a concrete example. Assume the recursive call worked on the concrete example. What do you have to do with that result to get the result you wanted for the initial call? Write code that uses the result in building the final result for your concrete example. By considering a concrete example it will help you see what computation is required to get your final result.*’
8. ‘*That’s it! Your function is complete and it will work if you stuck to these guidelines.*’

Recursion

Example

The previous functions are recursive functions.

```
factorial :: Int -> Int
length    :: [a] -> Int
(++)      :: [a] -> [a] -> [a]
last      :: [a] -> a
init      :: [a] -> [a]
null      :: [a] -> Bool
```

Characters and Strings

In Haskell the type of characters is `Char` and the type `String` is a type synonymous of `[Char]`. That is, a string is a list of characters.

Characters and Strings

In Haskell the type of characters is `Char` and the type `String` is a type synonymous of `[Char]`. That is, a string is a list of characters.

Example

```
’a’                        -- Character.
’a’ : ’b’ : ’c’ : []  -- List of characters.
[’a’,’b’,’c’]           -- List of characters.
"abc"                      -- String.

-- List of strings.
["hello","how"] ++ ["are","you?"]
```

Lazy Evaluation

Nothing is evaluated until necessary.

Lazy Evaluation

Nothing is evaluated until necessary.

Example (also in other programming languages)

```
-- Boolean disjunction.
(||) :: Bool -> Bool -> Bool
```

Lazy Evaluation

Example

```
foo :: Int -> Bool  -- Non-terminating function.
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n
```

Lazy Evaluation

Example

```
foo :: Int -> Bool  -- Non-terminating function.
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n
```

Question

Which is the value of `bar 10`?

Lazy Evaluation

Example

```
foo :: Int -> Bool  -- Non-terminating function.
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n
```

Question

Which is the value of `bar 10`?

```
GHCi> bar 10
True
```

Lazy Evaluation

Example (from `http://stackoverflow.com/questions/30688558/`)

```
dh :: Int -> Int -> (Int, Int)
dh d q = (2^d, q^d)

a :: (Int, Int)
a = dh 2 (fst b)

b :: (Int, Int)
b = dh 3 (fst a)
```

Lazy Evaluation

Example (from `http://stackoverflow.com/questions/30688558/`)

```
dh :: Int -> Int -> (Int, Int)
dh d q = (2^d, q^d)

a :: (Int, Int)
a = dh 2 (fst b)

b :: (Int, Int)
b = dh 3 (fst a)
```

Question

Which is the value of a?

Lazy Evaluation

Example (from `http://stackoverflow.com/questions/30688558/`)

```
dh :: Int -> Int -> (Int, Int)
dh d q = (2^d, q^d)

a :: (Int, Int)
a = dh 2 (fst b)

b :: (Int, Int)
b = dh 3 (fst a)
```

Question

Which is the value of a?

```
GHCi> a
(4,64)
```

Lazy Evaluation

Example

The expression `take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`.

```
take :: Int -> [a] -> [a]
```

Unbounded list.

```
ones :: [Int]
ones = 1 : ones
```

Lazy Evaluation

Example

The expression `take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`.

```
take :: Int -> [a] -> [a]
```

Unbounded list.

```
ones :: [Int]
ones = 1 : ones
```

Question

Which is the value of `take 5 ones`?

Lazy Evaluation

Example

The expression `take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n > length xs`.

```
take :: Int -> [a] -> [a]
```

Unbounded list.

```
ones :: [Int]
ones = 1 : ones
```

Question

Which is the value of `take 5 ones`?

```
GHCi> take 5 ones
[1,1,1,1,1]
```

Algebraic Data Types and Pattern Matching

Example

```haskell
data Bool = True | False
```

`True` and `False` are the (data) constructors for the data type `Bool`.

Algebraic Data Types and Pattern Matching

Example

```haskell
data Bool = True | False
```

`True` and `False` are the (data) constructors for the data type `Bool`.

Example (function by pattern matching)

```haskell
(||) :: Bool -> Bool -> Bool
True  || _ = True
False || x = x
```

Algebraic Data Types and Pattern Matching

Example

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Function by pattern matching.

```
nextDay :: Day -> Day
nextDay Mon = Tue
nextDay Tue = Wed
nextDay Wed = Thu
nextDay Thu = Fri
nextDay Fri = Sat
nextDay Sat = Sun
nextDay Sun = Mon
```

Algebraic Data Types and Pattern Matching

Example (recursive data type)

```
data Nat = Zero | Succ Nat
```

Algebraic Data Types and Pattern Matching

Example (recursive data type)

```
data Nat = Zero | Succ Nat
```

Example (structural recursive function by pattern matching)

```
(+) :: Nat -> Nat -> Nat
Zero     + n = n
(Succ m) + n = Succ (m + n)
```

Algebraic Data Types and Pattern Matching

Example (polymorphic data type)

```
data List a = Nil | Cons a (List a)
```

Algebraic Data Types and Pattern Matching

Example (polymorphic data type)

```haskell
data List a = Nil | Cons a (List a)
```

```haskell
-- The built-in lists.
data [] a = [] | a : [a]
```

Tuples

Example

See file `fp/Tuples.hs`.

Let Expressions and Where Clauses

Example

See file `fp/LetWhere.hs`.

Let Expressions and Where Clauses

Example

See file `fp/LetWhere.hs`.

Example

From [Hudak, Peterson and Fasel 1999].

```
let y   = a * b
    f x = (x + y)/y
in f c + f d
```

- The bindings created by a `let` expression are mutually recursive.
- The declarations permitted in `let` expressions include type signatures and function bindings.

Let Expressions and Where Clauses

Example

From [Hudak, Peterson and Fasel 1999].

```
f x y  |  y > z   =  ...
       |  y == z  =  ...
       |  y < z   =  ...
  where z = x * x
```

- A `where` clause is part of the syntax of function declarations.
- In this case, we cannot replace the `where` clause by a `let` expression.

Efficiency of Recursion

Example (Fibonnaci function)

Very inefficient version.

```
fib :: Natural -> Natural
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) +
        fib (n - 2)
```

The number of calls to `fib` grows exponentially with the size of `n`.

`fib 4`: 9 calls

`fib 5`: 15 calls

`fib 6`: 15 + 9 calls

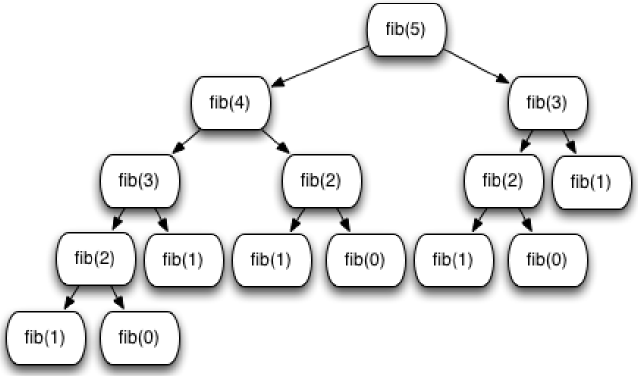

Fig. 5.16.

(continued on next slide)

Efficiency of Recursion

Example (continuation)

Accumulator pattern version.

```
fibAP :: Natural -> Natural
fibAP n =
  let fibH :: Natural -> Natural -> Natural -> Natural
      fibH count current previous =
        if count == n then previous
        else fibH (count + 1) (current + previous) current
  in  fibH 0 1 0
```

(continued on next slide)

Efficiency of Recursion

Example (continuation)

```
fibAP 5 = fibH 0 1 0
        = fibH 1 1 1
        = fibH 2 2 1
        = fibH 3 3 2
        = fibH 4 5 3
        = fibH 5 8 5
        = 5
```

$$\mathrm{fib}(0) = 0,$$
$$\mathrm{fib}(1) = 1,$$
$$\mathrm{fib}(2) = 1,$$
$$\mathrm{fib}(3) = 2,$$
$$\mathrm{fib}(4) = 3,$$
$$\mathrm{fib}(5) = 5.$$

(continued on next slide)

Efficiency of Recursion

Example (continuation)

Time running `fib 42` (file `fp/Fibonacci.hs`).

```
$ time ./fibonacci
real      1m4.353s
user      1m4.160s
sys       0m0.192s
```

Time running `fibAP 42` (file `fp/Fibonacci.hs`).

```
$ time ./fibonacci
real      0m0.006s
user      0m0.001s
sys       0m0.005s
```

Efficiency of Recursion

Example

Reverse of a list using append.

```
reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

Efficiency of Recursion

Example

Reverse of a list using append.

```
reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

Reverse of a list using the accumulator pattern.

```
reverse :: [a] -> [a]
reverse xs = rev xs []
  where
  rev []       zs = zs
  rev (y : ys) zs = rev ys (y : zs)
```

See file `fp/Reverse.hs`.

Tail Recursion

Problem

Recursive function calls takes longer than executing a simple loop.

Tail Recursion

Problem

Recursive function calls takes longer than executing a simple loop.

Solution

Tail recursion optimisation: Implement tail recursive functions using jump or branching instructions.

Tail Recursion

Problem

Recursive function calls takes longer than executing a simple loop.

Solution

Tail recursion optimisation: Implement tail recursive functions using jump or branching instructions.

Definition

> *'A **tail recursive function** is a function where the very last operation of the function is the recursive call to itself.' (p. 203)*

Tail Recursion

Problem

Recursive function calls takes longer than executing a simple loop.

Solution

Tail recursion optimisation: Implement tail recursive functions using jump or branching instructions.

Definition

> *‘A **tail recursive function** is a function where the very last operation of the function is the recursive call to itself.’ (p. 203)*

Example (Factorial function)

See directory `fp/factorial`.

Anonymous Functions

Example

The anonymous function $\lambda xy.y^2 + x$ can be represented in Hakell by

```
\ x y -> y * y + x
```

We applied the anonymous functions as usual

```
(\ x y -> y * y + x) 3 4
```

We also can bind an identifier to the anonymous function

```
foo :: Int -> Int -> Int
foo = (\ x y -> y * y + x)
foo 3 4
```

Higher-Order Functions

Definition

A function is **higher-order** iff

i) it takes a function as a parameter or

ii) it returns a function as its result.

Higher-Order Functions

Example

The composition operator `(.)` composes two functions. It is defined in the base library by

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \ x -> f (g x)
```

See file `fp/HigherOrder.hs`.

Higher-Order Functions

Example

The `map` functions applies a function to every element of a list.

The expression `map f xs` is the list obtained by applying `f` to each element of `xs`.

The `map` function is defined in the base library by

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs
```

See file `fp/HigherOrder.hs`.

Higher-Order Functions

Example

The `foldr` function (on lists) reduces a list using a binary operator from right to left, i.e.

```
foldr f z [x1, x2, ..., xn] ==
  x1 ‘f‘ (x2 ‘f‘ ... (xn ‘f‘ z)...)
```

The `foldr` function on lists can be defined by

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

See file `fp/HigherOrder.hs`.

Higher-Order Functions

Example

The `foldl` function (on lists) reduces a list using a binary operator from left to right, i.e.

```
foldl f z [x1, x2, ..., xn] ==
  (...((z ‘f‘ x1) ‘f‘ x2) ‘f‘...) ‘f‘ xn
```

The `foldl` function on lists can be defined by

```
foldl :: (a -> b -> b) -> b -> [a] -> b
foldl f z []       = z
foldl f z (x : xs) = foldl f (f z x) xs
```

Higher-Order Functions

Example

The `filter` function returns those elements of a list that satisfy a predicate (i.e., a function `a -> Bool`).

The `filter` function is defined in the base library by

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []       = []
filter p (x : xs)
  | p x           = x : filter p xs
  | otherwise     = filter p xs
```

Type Classes

Example

Does the element occur in the list?

```
elem :: a -> [a] -> Bool
elem x []        = False
elem x (y : ys) = x == y || elem x ys
```

Type Classes

Example

Does the element occur in the list?

```
elem :: a -> [a] -> Bool
elem x []        = False
elem x (y : ys) = x == y || elem x ys
```

The above code generates the following error:

```
error: No instance for (Eq a) arising from a use of ‘==’
```

(continued on next slide)

Type Classes

Example (continuation)

We can fix the error by adding the **type constraint** `Eq a` which restricts the type a to instances of the type class `Eq`.

```
elem :: Eq a => a -> [a] -> Bool
elem x []        = False
elem x (y : ys) = x == y || elem x ys
```

Type Classes

Description

Type classes provide a structured way to control ad hoc polymorphism, or overloading.

Example

The **type class** `Eq` is defined by

```
class Eq a where
  (==) :: a -> a -> Bool
```

Type Classes

Example

The data types `Bool` and `Nat` are **instances** of the type class `Eq`.

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

instance Eq Nat where
  Zero     == Zero     = True
  Zero     == (Succ _) = False
  (Succ _) == Zero     = False
  (Succ m) == (Succ n) = m == n
```

Type Classes

Example

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
GHCi> elem Mon [Tue, Sat, Sun]

error: No instance for (Eq Day) arising from a use of ‘==’
```

(continued on next slide)

Type Classes

Example (continuation)

A solution: Adding the missing instance

```
instance Eq Day where
  Mon == Mon = True
  Tue == Tue = True
  Wed == Wed = True
  Thu == Thu = True
  Fri == Fri = True
  Sat == Sat = True
  Sun == Sun = True
  _   == _   = False
```

(continued on next slide)

Type Classes

Example (continuation)

A solution: Using the deriving mechanism

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
         deriving Eq
```

Continuation Passing Style

Description

> *‘Continuation Passing Style (or CPS) is a way of writing functional programs where control is made explicit. In other words, the continuation represents the remaining work to be done.’ (p. 212)*

Continuation Passing Style

Description

> *'Continuation Passing Style (or CPS) is a way of writing functional programs where control is made explicit. In other words, the continuation represents the remaining work to be done.' (p. 212)*

Example

See file `fp/CPS.hs`.

Input and Output

The problem

How can programs with input and output be modelled as pure functions?

Input and Output

The problem

How can programs with input and output be modelled as pure functions?

The solution

There are various approaches for using pure functions and side-effects (see, e.g. [Peyton Jones and Wadler 1993]). Haskell's solution is via *monads*.

Input and Output

The unit type

The unit type is a type with only one element. Haskell unity type and its element are

```
() :: ()
```

The unit type is useful when performing input-output.

Input and Output

The `IO` type

The following description of the type `IO` is from [Hutton 2016, § 10.2].

Input and Output

The `IO` type

The following description of the type `IO` is from [Hutton 2016, § 10.2].

A program with input-output can be represented by a function

```
World -> World


type IO = World -> World
```

Input and Output

The `IO` type

The following description of the type `IO` is from [Hutton 2016, § 10.2].

A program with input-output can be represented by a function

```
World -> World


type IO = World -> World
```

What about if the program returns a value?

```
type IO a = World -> (a, World)
```

(continued on next slide)

Input and Output

The `IO` type (continuation)

What about if the program requires an argument?

For example, a program requiring a character and returning an integer has the type

```
Char -> IO Int


Char -> World -> (Int, World)
```

Input and Output

The `IO` type (continuation)

What about if the program requires an argument?

For example, a program requiring a character and returning an integer has the type

```
Char -> IO Int


Char -> World -> (Int, World)
```

The compiler has the responsibility of handling the state of world. The type `IO a` is primitive in Haskell.

Input and Output

Definition

An **action** is an expression of type `IO a`. When the expression is evaluated the action is performed.

Input and Output

Definition

An **action** is an expression of type `IO a`. When the expression is evaluated the action is performed.

Example

- `t : IO Char` is the action that returns a character.
- `t : IO ()` is the action that no returns a value, where `()` is a dummy result value.

Input and Output

The abstract datatype `IO a`

The abstract datatype `IO a` has (at least) the following operations [Bird 1998, § 10.1]:

```
return  :: a -> IO a
(>>=)   :: IO a -> (a -> IO b) -> IO b
putChar :: Char -> IO ()
getChar :: IO Char
```

Input and Output

Example

See file `fp/IO.hs`.

Testing via QuickCheck

A paper

Claessen, Koen and Hughes, John [2000]. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP'00. DOI: `https://doi.org/10.1145/357766.351266`.

---

[*] See `www.sigplan.org/Awards/ICFP/`.

Testing via QuickCheck

A paper

Claessen, Koen and Hughes, John [2000]. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP'00. DOI: `https://doi.org/10.1145/357766.351266`.

Most Influential ICFP Paper Award 2010*

> *'The techniques described in the paper have spawned a significant body of follow-on work in test case generation. They have also been adapted to other languages . . . '*

---

*See `www.sigplan.org/Awards/ICFP/`.

Testing via QuickCheck

An open source library

QuickCheck on Hackage.*

---

* `http://hackage.haskell.org/package/QuickCheck`.

Testing via QuickCheck

An open source library

QuickCheck on Hackage.[*]

Commercialisation

QuviQ (`www.quviq.com/`).

---

[*] `http://hackage.haskell.org/package/QuickCheck`.

Testing via QuickCheck

Adaptations

QuickCheck has been ported to various languages (Wikipedia 2023-10-17).

- C
- C#
- C++
- Chicken
- Clojure
- Common Lisp
- Coq
- D
- Elm
- Elixir
- Erlang
- F#
- Factor
- Go
- Io
- Java
- JavaScript
- Julia
- Logtalk
- Lua
- Mathematica
- Objective-C
- OCaml
- Perl
- Prolog
- PHP
- Pony
- Python
- R
- Racket
- Ruby
- Rust
- Scala
- Scheme
- Smalltalk
- Standard ML
- Swift
- TypeScript
- Visual Basic .NET
- Vhiley

Testing via QuickCheck

False positive

The program works properly but the test pointed out a fail:

- There is a bug elsewhere.
- There is an error in the specification.

Testing via QuickCheck

False positive

The program works properly but the test pointed out a fail:

- There is a bug elsewhere.
- There is an error in the specification.

False negative

There is a bug in the program but the test passed.

Recall Dijkstra's 1969 famous quote:

> *'Testing shows the presence, not the absence of bugs.' (Buxton and Randell 1970)*

Testing via QuickCheck

QuickCheck demo

References

Barendregt, H. P. [1981] (2004). The Lambda Calculus. Its Syntax and Semantics. Revised edition, 6th impression. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier (cit. on p. 16).

Bird, Richard [1988] (1998). Introduction to Functional Programming. 2nd ed. Prentice Hall Press (cit. on p. 111).

Buxton, J. N. and Randell, B., eds. (1970). Software Engineering Techniques (NATO Software Engineering Conference 1969). (Cit. on pp. 118, 119).

Church, Alonzo (1932). A Set of Postulates for the Foundation of Logic. Annals of Mathematics 33.2, pp. 346–366. DOI: `10.2307/1968337` (cit. on pp. 8–10).

— (1935). An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5, pp. 332–333. DOI: `10.1090/S0002-9904-1935-06102-6` (cit. on p. 23).

— (1936). An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2, pp. 345–363. DOI: `10.2307/2371045` (cit. on p. 23).

References

Hindley, J. Roger and Seldin, Jonathan P. (2008). Lambda-Calculus and Combinators. An Introduction. Cambridge University Press (cit. on p. 16).

Hudak, Paul, Hughes, John, Peyton Jones, Simon and Wadler, Philip (2007). A History of Haskell: Being Lazy with Class. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. HOPL III, 12:1–12:55. DOI: `10.1145/1238844.1238856` (cit. on p. 24).

Hudak, Paul, Peterson, John and Fasel, Joseph H. (1999). A Gentle Introduction to Haskell 98. URL: `https://www.haskell.org/tutorial/` (cit. on pp. 71–73).

Hughes, J. (1989). Why Functional Programming Matters. The Computer Journal 32.2, pp. 98–107. DOI: `10.1093/comjnl/32.2.98` (cit. on pp. 27, 28).

Hutton, Graham [2007] (2016). Programming in Haskell. 2nd ed. Cambridge University Press (cit. on pp. 4–6, 104–106).

Lee, Kent D. [2014] (2017). Foundations of Programming Languages. 2nd ed. Undergraduate Topics in Computer Science. Springer (cit. on p. 2).

O'Sullivan, Bryan, Goerzen, John and Stewart, Don (2008). Real World Haskell. O'Really Media, Inc. (cit. on pp. 4–6).

References

Peyton Jones, Simon L. and Wadler, Philip (1993). Imperative Functional Programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1993), pp. 71–84. DOI: 10.1145/158511.158524 (cit. on pp. 101, 102).