Functional Programming

(short overview)

Programming Techniques II HS15

λ-calculus

- Universal model of computation

  → study computable functions (≈algorithms)

- Simplifies mathematical formulas into λ-terms
- Basis for functional programming
- Typed and untyped
- Alternative to Turing Machines

λ-calculus (informal)

- Functions are anonymous and bind variables to λ-terms:

  $\text{square}(x) := x \cdot x \quad \Rightarrow \quad x \mapsto x \cdot x \qquad (\text{ or } \lambda x.x \cdot x)$

- Functions only have a single input:

  $x,y \mapsto ? \quad \Rightarrow$ currying (later)

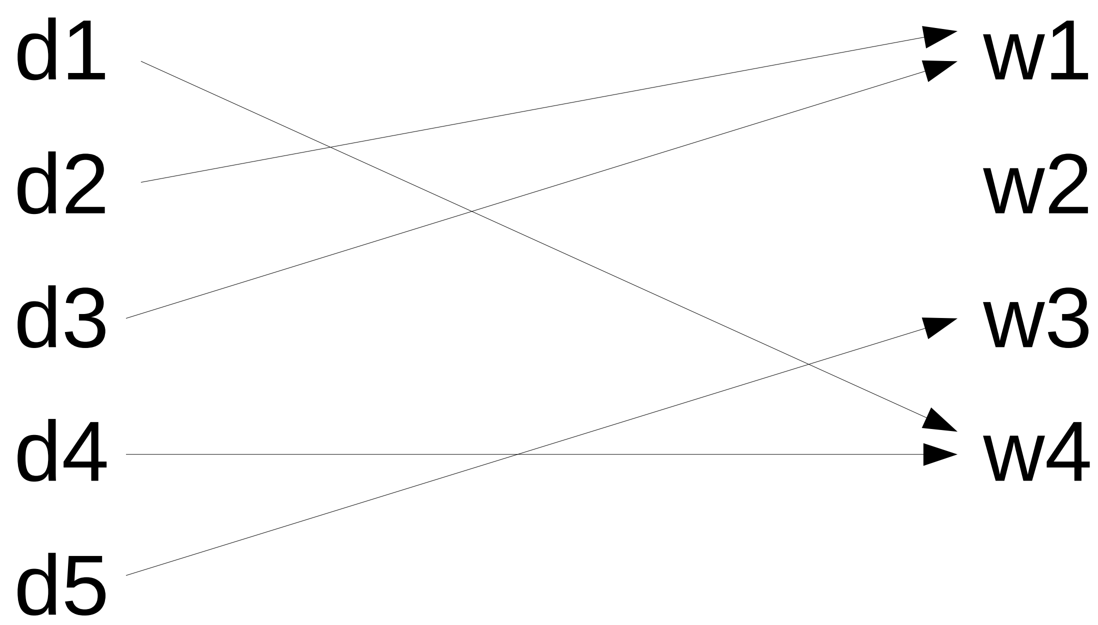
- Simple syntax (abstraction and application):

  $(x \mapsto \lambda_1) =: \lambda_2 \qquad \lambda_1\lambda_2\lambda_3 =: \lambda_4$

- Evaluation can be carried out in any order (associative)

Pure Functions

Domain $\mathcal{D}$ $\Longrightarrow$ Codomain $\mathcal{W}$



Mapping existing values to existing values.

A function changes nothing!

Pure Functions

A function call in a functional language can always be replaced by its implementation.

Returned description for `y = f(x)` is independent of context and time:

`assert(f(x) == f(x))` always true (for non-volatile x)

Contrast (Pseudo Code)

Functional

```
f(x) := x+1
```

```
y = f(2)
z = y + 1
```

→ z = $\underline{x+1}$+1 | x = 2

λ_f

Procedural & Object Oriented

```
f(x) := {
    x.add(1)
    return x
}
```

→ z = 3+1

FP is very, very lazy!

Assignment

The **assignment** statement induces state.

No assignmnent → no side effects, which may be required in non-pure applications:

- std::**ofstream**::open std::**ofstream**::close
- **`new delete`**
- `std::lock_guard<std::mutex>  ~lock_guard()`

Stateful functions: always two there are, and one the other follow must.

Yes, functional programmers have a problem with =

- No assignment in FP. But what then?

Haskell

- Named after Haskell Curry (1900-1982)

- No `for`, `while`, `goto`
- No assignment or even variables...
- But we have constants and functions!

Currying

- Our functions take a single argument
- Multiple arguments are chained functions -> currying:

  $(x,y) \mapsto x+y \quad => \quad x \mapsto (y \to x+y)$

```
g :: (x, y) -> z
f = curry g
f :: x -> y -> z
```

Haskell Output

```haskell
str = "What does this do"   -- constant
ask = (++ "?")              -- function
confusedly f x = f(f(f(x)))-- composition

main = print( confusedly ask str )

-- What does this do???
```

Declarative Programming

- *Imperative* programming is a step-by-step cooking recipe.
- We don't define steps to change objects. We define what objects should be!
- What about real changes like user input, disk I/O, etc.?

Monads (Monoids)

- Algebraic structure for calculation
- Modelled after monoids from group theory:

  For type $\mathcal{S}$, element e $\in \mathcal{S}$:

  - binary operations on e_1, e_2 are still $\in \mathcal{S}$
  - $\exists$ identity element
  - associativity

Monads (simple)

- Rules for *composition of expressions*
- Expand type with new rules: **`int`** can be *lifted* to a type supporting `NaN` after division by zero
- Must still work with other **`int`**s (*bind*)
- Must be transparent to algorithms (*unit / return*)

Monads (simple)

- The Input/Output monad represents the world:
  - deleting a file *describes* an action which upon execution ensures the file doesn't exist

```
world_without_XYZ = removeFile(xyz)
do_work(world_without_XYZ)


main :: IO ()
```

- Associativity of description allows for stateless programms!

Core Principles

- Stateless
- No side effects
- Functions are first class citizens
- Types
- Composition

Properties

- Lazy evaluation
- Parameters and return values are functions
- Higher order functions are common
- Recursion
- No memory, no caching
- Modularity (everything behaves like concepts)

Recursion

We have no side effects and immutable data, we can not loop over an array and modify its values.

→ algorithms have to be recursive

→ types (not classes) without state/privates

Factorial

```
factorial n | n < 2 = 1
factorial n = n * factorial (n - 1)

main = print(factorial 3210) --9865 digits
```

Factorial

```
factorial n
  | n < 2       = 1                 -- guards
  | otherwise = n * factorial (n - 1)

main = print(factorial 3210) --9865 digits
```

Factorial

```haskell
factorial n = product [1..n]

main = print(factorial 3210) --9865 digits
```

Factorial

```
factorial :: Integer -> Integer
factorial n = product [1..n]

main = print(factorial 3210) --9865 digits
```

Cost of Recursion

- Memory is cheap!
- But memory access can be slow

Tail Call

```
int foo(int x) {
  if(x > 5)
    return bar(x);    // tail call
  else
    return bar(x)+1;  // not tail call
}
```

main() → foo(x) → bar(x)

with x==5: main() → bar(x)

Tail Call Recursion

```
int bar(int x) {
    if(x == 1) return 1;
    return bar(x-1);
}
```

backtrace if(x==1):

bar(int)
bar(int)
bar(int)
bar(int)
bar(int)
foo(int)
main()

bar(5);
bar(bar(4));
bar(bar(bar(3)));
bar(bar(bar(bar(2))));
bar(bar(bar(bar(bar(1)))));
bar(bar(bar(bar(1))));
bar(bar(bar(1)));
bar(bar(1));
bar(1);
1;

Tail Call Elimination

gcc/clang with at least -O2:

```
int bar(int x) {
LBL:
    if(x == 1) return 1;
    --x;
    goto LBL;
}
```

bar(5);

bar(4);

bar(3);

bar(2);

bar(1);

1;

backtrace if(x==1):

bar(int)
foo(int)
main()

Tail Call Elimination

Note: the return statement must be a function call only, not an expression.

In pseudo-assembler:

```
bar:
    call B
    call A
    ret
```

```
bar:
    call B
    jmp A
```

Quicksort

```
quicksort :: (Ord a) => [a] -> [a]  -- lift comparable type
Quicksort [] = []                   -- empty list is sorted
quicksort (x:xs) = quicksort [y | y <- xs, y < x ]
                   ++ [x]           -- pivot is first elem
                   ++ quicksort [y | y <- xs, y >= x]

main = print(quicksort [2, 5, 1, 3, 2, 1, 9])
```

Data

We have no concept of object state nor memory (heap).

→ compose types, not classes (no instances, no privates)

→ types need to be enabled through monads

OO vs FP

- Design Patterns
- Inheritance
- Classes & types
- Exposed procedure, hidden state
- Data access freedom

- Functions
- Function composition/monads
- Types (functions)
- Only procedure, no state
- Safety

OO vs FP

“Object-oriented programming is an exceptionally bad idea which could have originated in California”

- E.W. Dijkstra

“You probably know that arrogance, in computer science, is measured in nanodijkstras.”

- Alan Kay

Clojure

Functional* language based on LISP

- Code and data have same (simple!) syntax
- Implemented on JVM
- Transactions for change of state

*slightly impure

Clojure Syntax

Syntax:

```
(print "Hello world!")
(println)
```

Clojure:

```
(defn hi []
      "Hello world!")
```

C++:

```
std::string hi() {
  return “Hello world!”;
}
```

Clojure Functions

```
(defn hi [] "Hello world!")
```

```
(hi)                  ; prints nothing
(println hi)          ; prints type of function hi
(println (hi))        ; prints "Hello world!"
```

Clojure Functions

```
(defn square [x]
  (* x x))
(let [x 2 y 3]
  (println (square (+ x y))))
```

Prints:

25

```
(let [x [0 1 2]]
    (let [x (conj x 3)]
        (println x))
    (println x))
```

Prints:

[0 1 2 3]

[0 1 2]

Clojure Functions

```
; Composition
((comp inc *) 3 3)
```

Prints:

10

```
; Lots of sugar
(reduce + (range 1 10 2))
```

Prints:

25

Threading

- Moore's law today: # of cores
- Threading on multiple cores: little hand-holding, race conditions, synchronisation

- `assert(f(x) == f(x))`

  on two cores will always hold for pure `f`

Threading

```
(defn f1 [] (prn "f1"))
(defn f2 [] (prn "f2"))

(pvalues (f1) (f2))
```

FP Languages

Haskell	Scala
Clojure	F#
Erlang	Elixir
C++ MTP	OCaml
C++11 constexpr functions	ML

FP Languages

Haskell

Clojure

Erlang

C++ MTP

C++11 constexpr functions

Scala

F#

ML

- purely functional
- widely studied in academia
- Bank of America, Deutsche Bank, ...
- hundreds of companies and projects

FP Languages

Haskell

Clojure

Erlang

C++ MTP

C++11 constexpr functions

Scala

F#

ML

- LISP heritage
- big data handling
- used in dozens of projects

FP Languages

Haskell

Clojure

Erlang

- garbage collected, highly concurrent
- Facebook chat backend
- WhatsApp messaging servers

C++ MTP

C++11 constexpr functions

Scala

F#

Elixir

FP Languages

Haskell	Scala
Clojure	F#
Erlang	
C++ MTP	OCaml
C++11 constexpr functions	ML

- compile time computation
- used extensively in Boost

FP Languages

Haskell

Clojure

Erlang

C++ MTP

C++11 constexpr functions

Scala

F#

Elixir

OCaml

ML

- PT2 exercises!

FP Languages

Haskell	Scala
Clojure	F#
Erlang	Elixir
C++ MTP	OCaml
C++11 constexpr functions	ML

FP is Awesome

- Beautiful and elegant (few lines of code)
- Debugging is wonderful ("What's the state?" - "I don't care!")
- Peace of mind (“Did I remember to deallocate?”)
- Separation of concerns (parallelisation, convenient threading)
- Performance through immutability (const) and lazy evaluation

FP is Awesome?

Well...

- It's not exactly popular among the folks
- Genuinely harder to write
- Implementing stateful optimisation techniques is a crutch (e.g. runtime caching, even though there's memoize)
- Tail call elimination not guaranteed in certain cases (→ recur, trampoline)