Theory of Computer Science: Why All That Formal Stuff?

Sergiu Ivanov

sergiu.ivanov@univ-grenoble-alpes.fr

Université Grenoble Alpes

Open Seminar

Question

Computer Science $\leftrightarrow$ Maths

What is the relationship?

Outline

Outline

Calculus

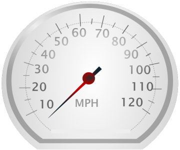
derivatives $\frac{df}{dx}$ integrals $\int_a^b f\,dx$

https://openclipart.org/

Calculus

derivatives $\dfrac{df}{dx}$

integrals $\displaystyle\int_a^b f\,dx$

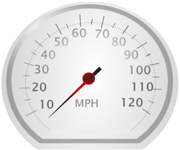


https://openclipart.org/

Calculus

derivatives $\dfrac{df}{dx}$

integrals $\displaystyle\int_a^b f\,dx$

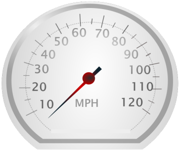


How often do we use that in practice?

https://openclipart.org/

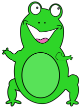
Calculus

derivatives $\frac{df}{dx}$ integrals $\int_a^b f\,dx$



How often do we use that in practice?

We use that in games!



collisions, ray tracing, ...

https://openclipart.org/

Outline

Formal Languages

Finite alphabet: $V = \{a_1, a_2, \ldots, a_n\}$ ← letters

Formal Languages

Finite alphabet: $V = \{a_1, a_2, \dots, a_n\}$ letters

Word = any finite sequence of letters

- $a_1 a_2$, $a_1 a_1 a_1$, $a_2 a_2 a_1 a_1 a_2 a_2$

Formal Languages

Finite alphabet: $V = \{a_1, a_2, \ldots, a_n\}$ letters

Word = any finite sequence of letters

- a_1a_2, $a_1a_1a_1$, $a_2a_2a_1a_1a_2a_2$

Language over V = any set of words over V

Formal Languages

Finite alphabet: $V = \{a_1, a_2, \ldots, a_n\}$ letters

Word = any finite sequence of letters

- a_1a_2, $a_1a_1a_1$, $a_2a_2a_1a_1a_2a_2$

Language over V = any set of words over V

regular languages, finite automata, pushdown automata, Turing machines, context-free language, pumping lemma, ...

Formal Languages

Finite alphabet: $V = \{a_1, a_2, \ldots, a_n\}$ letters

Word = any finite sequence of letters

- a_1a_2, $a_1a_1a_1$, $a_2a_2a_1a_1a_2a_2$

Language over V = any set of words over V

regular languages, finite automata, pushdown automata, Turing machines, context-free language, pumping lemma, ...

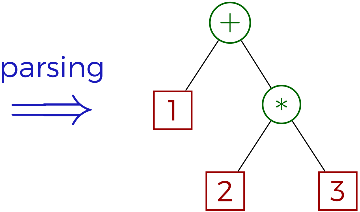
Why? Why?

Formal Languages: Compilers

Programming languages are formal languages

- alphabet for C $= \{\texttt{if}, \texttt{for}, \texttt{int}, \texttt{+}, \texttt{*}, \ldots\}$

$1 + 2 * 3$

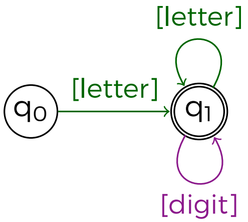


Compiler $=$ parser $+$ binary code generator

Formal Languages: Regular Expressions

[letter]([letter] | [digit])*

- a, ab, c2, x2a, ...



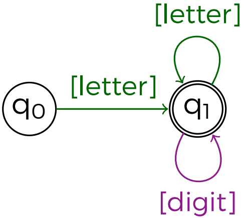
finite automaton

Formal regular expressions $\sim$ finite automata

Formal Languages: Regular Expressions

$[\text{letter}]\Big([\text{letter}] \,\Big|\, [\text{digit}]\Big)^*$

- a, ab, c2, x2a, ...



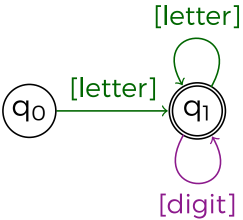
finite automaton

Formal regular expressions $\sim$ finite automata

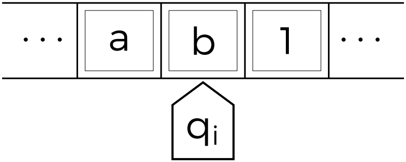
Regexp $=$ rather extended regular expressions

Formal Languages: A Philosophy of Computers

Finite automata



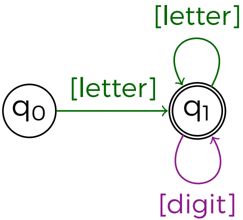
Turing machines

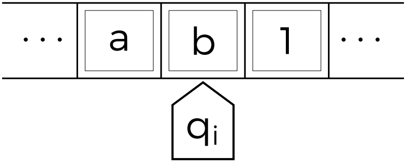


Formal Languages: A Philosophy of Computers

strictly less powerful

Finite automata $<$ Turing machines



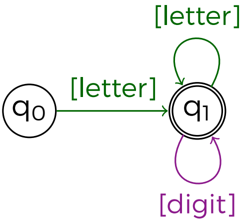


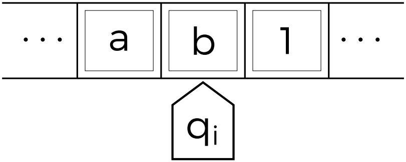
Formal Languages: A Philosophy of Computers

strictly less powerful

↓

Finite automata < Turing machines



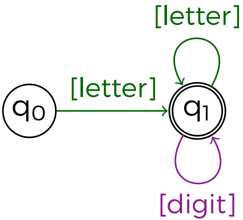


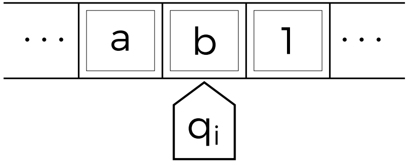
Computers correspond to which?

Formal Languages: A Philosophy of Computers

strictly less powerful

Finite automata $<$ Turing machines





Computers correspond to which?

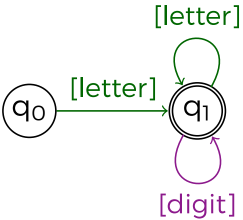
Finite automata!

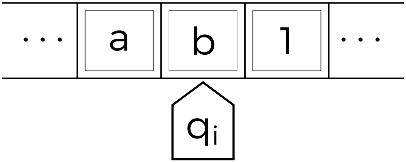
- all resources are finite

Formal Languages: A Philosophy of Computers

strictly less powerful

Finite automata < Turing machines





Computers correspond to which?

Finite automata!

- all resources are finite

but

Programming languages are Turing powerful!

Outline

Set Theory

$$A = \{a, b, c, \dots\}$$

When do programmers use set theory?

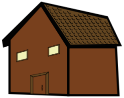
Classes and Types “are” Sets

A class/type is a set of objects sharing a property.

https://openclipart.org/

Classes and Types "are" Sets

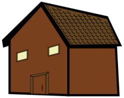
A class/type is a set of objects sharing a property.

house = {  ,  ,  , . . . }

https://openclipart.org/

Classes and Types “are” Sets

A class/type is a set of objects sharing a property.

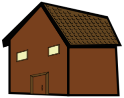
house = {  ,  ,  , . . . }

Inheritance = set inclusion

https://openclipart.org/

Classes and Types “are” Sets

A class/type is a set of objects sharing a property.

house = {  ,  ,  , . . . }

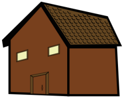
Inheritance = set inclusion

$\text{house} \subseteq \text{building}$

https://openclipart.org/

Classes and Types “are” Sets

A class/type is a set of objects sharing a property.

house = {  ,  ,  , . . . }

Inheritance = set inclusion

$\text{house} \subseteq \text{building}$

Every house is a building, but not every building is a house.

`https://openclipart.org/`

Types and Operations on Sets

`MyType x;` $\quad$ $x \in \text{MyType}$

Types and Operations on Sets

```
MyType x;
```

$x \in \text{MyType}$

```
struct Person {
  String name;
  int age;
}
```

$\text{Person} = \text{String} \times \text{int} = \{(\text{“Vasile”}, 1234), (\text{“Ion”}, -2), \ldots\}$

Types and Operations on Sets

```
MyType x;
```

$x \in \text{MyType}$

```
struct Person {
  String name;
  int age;
}
```

$\text{Person} = \text{String} \times \text{int} =$
$\{(\text{“Vasile”}, 1234), (\text{“Ion”}, -2), \dots\}$

$\text{Person} = \text{String} \cup \text{int} =$
$\{\text{“Vasile”}, 1234, \text{“Ion”}, -2, \dots\}$

Types and Operations on Sets

```
MyType x;
```

$x \in \text{MyType}$

```
struct Person {
  String name;
  int age;
}
```

$\text{Person} = \text{String} \times \text{int} = \{(\text{“Vasile”}, 1234), (\text{“Ion”}, -2), \dots\}$

```
union Variant {
  String str;
  int num;
}
```

$\text{Person} = \text{String} \cup \text{int} = \{\text{“Vasile”}, 1234, \text{“Ion”}, -2, \dots\}$

Types and Operations on Sets

```
MyType x;
```

$x \in \text{MyType}$

```
struct Person {
  String name;
  int age;
}
```

$\text{Person} = \text{String} \times \text{int} = \{(\text{“Vasile”}, 1234), (\text{“Ion”}, -2), \ldots\}$

```
union Variant {
  String str;
  int num;
}
```

$\text{Person} = \text{String} \cup \text{int} = \{\text{“Vasile”}, 1234, \text{“Ion”}, -2, \ldots\}$

```
f :: Int -> Double
f x = x / 2
```

$f : \mathbb{Z} \to \mathbb{R} \in \mathbb{R}^{\mathbb{Z}}$

Types and Operations on Sets

```
MyType x;
```

$x \in \text{MyType}$

```
struct Person {
  String name;
  int age;
}
```

$\text{Person} = \text{String} \times \text{int} = \{(\text{“Vasile”}, 1234), (\text{“Ion”}, -2), \ldots\}$

```
union Variant {
  String str;
  int num;
}
```

$\text{Person} = \text{String} \cup \text{int} = \{\text{“Vasile”}, 1234, \text{“Ion”}, -2, \ldots\}$

```
f :: Int -> Double
f x = x / 2
```

$f : \mathbb{Z} \to \mathbb{R} \in \mathbb{R}^{\mathbb{Z}}$

How about $\cap$, $\setminus$, ... ?

https://openclipart.org/

Outline

Abstract Algebra

Group 

https://openclipart.org/

Abstract Algebra

Group

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$
- inverses: $x + (-x) = (-x) + x = 0$

https://openclipart.org/

Abstract Algebra

Group

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$
- inverses: $x + (-x) = (-x) + x = 0$

Free Monoid

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$

https://openclipart.org/

Abstract Algebra

Group 

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$
- inverses: $x + (-x) = (-x) + x = 0$

Free Monoid

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$

Who uses monoids?? 

https://openclipart.org/

Abstract Algebra

Group

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$
- inverses: $x + (-x) = (-x) + x = 0$

Free Monoid

- associativity: $x + (y + z) = (x + y) + z$
- identity: $x + 0 = 0 + x = x$

Who uses monoids??  Turns out, you do!

https://openclipart.org/

Monoids as Collections

In a free monoid M, no sum cancels out.

Take $x \in M$ → $x + 0 = x$, same length

Take $x \in M$ → $x + y$, a longer sum

Monoids as Collections

In a free monoid M, no sum cancels out.

Take $x \in M$ → $x + 0 = x$, same length
Take $x \in M$ → $x + y$, a longer sum

Formal sums in a free monoid represent collections.

The terms of the sum are the entries.

Monoids as Collections

In a free monoid M, no sum cancels out.

Take $x \in M$ →
- $x + 0 = x$, same length
- $x + y$, a longer sum

Formal sums in a free monoid represent collections.

The terms of the sum are the entries.

The sum operator is the concatenation.

- [1,3] + [3,7] = [1,3,3,7]
- "big" + "banana" = "bigbanana"

Monoids as Collections

In a free monoid M, no sum cancels out.

Take $x \in M$ → $x + 0 = x$, same length

Take $x \in M$ → $x + y$, a longer sum

Formal sums in a free monoid represent collections.

The terms of the sum are the entries.

The sum operator is the concatenation.

- [1,3] + [3,7] = [1,3,3,7]
- “big” + “banana” = “bigbanana”

A log is a typical free monoid.

Outline

“Easy” Parallelism with Functional Programming

Higher-order functions are easier to handle.

```
for(i = 0; i < n; i++)
  vect[i] = vect[i] + 2;
```

```
map (λ x → x + 2) vect
```

easier to parallelise

"Easy" Parallelism with Functional Programming

Higher-order functions are easier to handle.

```
for(i = 0; i < n; i++)
  vect[i] = vect[i] + 2;
```

map (λ x $\rightarrow$ x + 2) vect

easier to parallelise

- each step in `for` explicitly depends on the previous one: `i = i+1`
- the behaviour of `map` is explicitly fixed

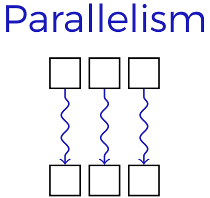
Parallelism vs. Concurrency. Statically.

Parallelism

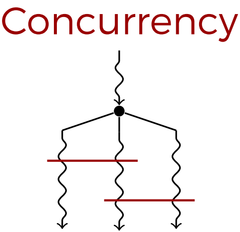
Concurrency

Parallelism vs. Concurrency. Statically.

1. problem → independent subproblems
2. solve subproblems independently
   - no shared resources



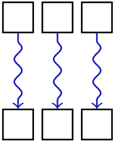
multiple threads share resources
- synchronisation



Parallelism vs. Concurrency. Statically.

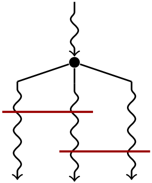
1. problem → independent subproblems
2. solve subproblems independently
    - no shared resources

Parallelism



multiple threads share resources
- synchronisation

Concurrency

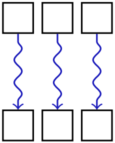


Types allow static differentiation between parallel threads and concurrent threads.

Parallelism vs. Concurrency. Statically.

1. problem → independent subproblems
2. solve subproblems independently
   - no shared resources

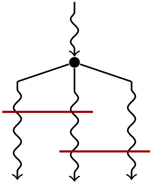
Parallelism



multiple threads share resources

- synchronisation

Concurrency



Types allow static differentiation between parallel threads and concurrent threads.

- monads

Outline

Successive Lookups

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

Successive Lookups

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

```
case personByName "John" of
  Nothing -> Nothing
  Just john ->
```

Successive Lookups

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

```
case personByName "John" of
  Nothing -> Nothing
  Just john ->
    case carByPerson john of
      Nothing -> Nothing
      Just johnsCar -> model johnsCar
```

Successive Lookups

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

```
case personByName "John" of
  Nothing -> Nothing
  Just john ->
    case carByPerson john of
      Nothing -> Nothing
      Just johnsCar -> model johnsCar
```

Imagine what happens if one has longer chains.

https://openclipart.org/

Factoring out Patterns (Monads! \o/)

We often want to do the same thing over and over between two function calls:

- check whether the previous lookup returned a value

Factoring out Patterns (Monads! \o/)

We often want to do the same thing over and over between two function calls:

- check whether the previous lookup returned a value
- handle states

Factoring out Patterns (Monads! \o/)

We often want to do the same thing over and over between two function calls:

- check whether the previous lookup returned a value
- handle states
- strictly specify and handle side effects

Factoring out Patterns (Monads! \o/)

We often want to do the same thing over and over between two function calls:

- check whether the previous lookup returned a value
- handle states
- strictly specify and handle side effects

Monads help factor out such patterns.

Conclusion

Thinking formally may be useful.

Conclusion

Thinking formally may be useful.

Don't overdo it tho.

- that's the subject of my next talk