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 ↔ Maths

What is the relationship?
Outline

1. Part 1
   - Calculus
   - Formal Languages
   - Set Theory

2. Part 2
   - Collections
   - Parallel and Concurrent Programming
   - Factoring Out Some Repeating Patterns
Outline

1. Part 1
   Calculus
   Formal Languages
   Set Theory

2. Part 2
   Collections
   Parallel and Concurrent Programming
   Factoring Out Some Repeating Patterns
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, …
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, …
Calculus

derivatives $\frac{df}{dx}$

integrals $\int_{a}^{b} f \, dx$

How often do we use that in practice?
How often do we use that in practice?

We use that in games! collisions, ray tracing, ...
Outline

1. Part 1
   Calculus
   Formal Languages
   Set Theory

2. Part 2
   Collections
   Parallel and Concurrent Programming
   Factoring Out Some Repeating Patterns
Formal Languages

Finite alphabet: \( V = \{a_1, a_2, \ldots, a_n\} \) letters
Formal Languages

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

Word = any finite sequence of letters

- \( a_1a_2, a_1a_1a_1, a_2a_2a_1a_1a_2a_2 \)
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, ...
Programming languages are formal languages

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

1 + 2 \times 3 \quad \xrightarrow{\text{parsing}} \quad 1 + 2 \times 3

Compiler = parser + binary code generator
Formal Languages: Regular Expressions

\[
\text{[letter]} \left( \text{[letter]} \mid \text{[digit]} \right)^* \\
\]

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

Formal regular expressions $\sim$ finite automata

finite automaton
Formal Languages: Regular Expressions

\[ [\text{letter}] \left( [\text{letter}] \mid [\text{digit}] \right)^* \]

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

Formal regular expressions \( \sim \) finite automata

\[ \text{Regexp} = \text{rather extended regular expressions} \]
Finite automata

Turing machines

Finite automata correspond to all resources being finite, but programming languages are Turing powerful!
Finite automata $\lessdot$ Turing machines

Formal Languages: A Philosophy of Computers

strictly less powerful

$q_0 \xrightarrow{\text{letter}} q_1 \xrightarrow{\text{digit}} q_i$

$q_0 \xrightarrow{\text{letter}} q_1$

$\cdots a b 1 \cdots$

Programming languages are Turing powerful!

Theoretical Computer Science: Why All That Formal Stuff?
Formal Languages: A Philosophy of Computers

strictly less powerful

Finite automata $<$ Turing machines

Computers correspond to which?
Formal Languages: A Philosophy of Computers

Finite automata \( \lessdot \) Turing machines

Finite automata!
- all resources are finite

Computers correspond to which?

Sergiu Ivanov, Université Grenoble Alpes
Formal Languages: A Philosophy of Computers

Finite automata \( \leq \) Turing machines

Computers correspond to which?

Finite automata! but Programming languages are Turing powerful!

- all resources are finite
Outline

1. Part 1
   - Calculus
   - Formal Languages
   - Set Theory

2. Part 2
   - Collections
   - Parallel and Concurrent Programming
   - Factoring Out Some Repeating Patterns
Set Theory

\[ A = \{a, b, c, \ldots \} \]

When do programmers use set theory?
A **class/type** is a **set** of objects sharing a **property**.
Classes and Types “are” Sets

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

\[
\text{house} = \{ \text{house}, \text{building}, \ldots \}
\]
Classes and Types “are” Sets

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

\[
\text{house} = \{ \text{house}, \text{building}, \ldots \}
\]

Inheritance = set inclusion
Classes and Types “are” Sets

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

\[
\text{house} = \{ \text{house}, \text{house}, \text{house}, \ldots \}
\]

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.

\[
\text{house} = \{\text{house}, \text{building}, \ldots\}
\]

Inheritance = set inclusion

\[
\text{house} \subseteq \text{building}
\]

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

Person = String × int =
{(“Vasile”, 1234), (“Ion”, −2), …}
Types and Operations on Sets

MyType x;

\[ x \in \text{MyType} \]

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

\[ \text{Person} = \text{String} \times \text{int} = \{(\text{"Vasile"}, 1234), (\text{"Ion"}, -2), \ldots \} \]

```
Person = String \cup \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;
}

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

union Variant {
    String str;
    int num;
}

Person = String \cup int = \{(\text{"Vasile"}, 1234, \text{"Ion"}, -2, \ldots \}

How about \( n \), \( \ldots \)?
Types and Operations on Sets

MyType \( x; \quad x \in \text{MyType} \)

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

Person = String \( \times \) int = \{("Vasile", 1234), ("Ion", -2), \ldots \}

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

Person = String \( \cup \) int = \{"Vasile", 1234, "Ion", -2, \ldots \}

```
f :: Int \rightarrow \text{Double} 
f x = x / 2
```

\[ f : \mathbb{Z} \rightarrow \mathbb{R} \in \mathbb{R}^\mathbb{Z} \]
Types and Operations on Sets

\[
\begin{align*}
\text{MyType } x; & \quad x \in \text{MyType} \\
\text{struct Person} & \{ \\
\text{String name; } & \quad \text{Person} = \text{String} \times \text{int} = \\
\text{int age; } & \quad \{ ("Vasile", 1234), ("Ion", -2), \ldots \} \\
\} \\
\text{union Variant} & \{ \\
\text{String str; } & \quad \text{Person} = \text{String} \cup \text{int} = \\
\text{int num; } & \quad \{ "Vasile", 1234, "Ion", -2, \ldots \} \\
\} \\
f :: \text{Int} \to \text{Double} & \quad f : \mathbb{Z} \to \mathbb{R} \in \mathbb{R}^\mathbb{Z} \\
f x = x / 2
\end{align*}
\]

How about \( \cap, \setminus, \ldots \)?

https://openclipart.org/
Outline

1. Part 1
   - Calculus
   - Formal Languages
   - Set Theory

2. Part 2
   - Collections
   - Parallel and Concurrent Programming
   - Factoring Out Some Repeating Patterns
Abstract Algebra

Group

Abstract Algebra

Group

Who uses monoids? Turns out, you do!
Abstract Algebra

Group

- **associativity:** \( x + (y + z) = (x + y) + z \)
- **identity:** \( x + 0 = 0 + x = x \)
- **inverses:** \( x + (-x) = (-x) + x = 0 \)
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: \( x + 0 = 0 + x = x \)
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!
Monoids as Collections

In a free monoid $M$, no sum cancels out.

Take $x \in M$

\[ x + 0 = x, \text{ same length} \]

\[ x + y, \text{ a longer sum} \]
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.
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 $x \in M$

$\rightarrow x + 0 = x$, same length

$\rightarrow x + y$, 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.

$\rightarrow [1,3] + [3,7] = [1,3,3,7]

\rightarrow "big" + "banana" = "bigbanana"

A log is a typical free monoid.
Outline

1. **Part 1**
   - Calculus
   - Formal Languages
   - Set Theory

2. **Part 2**
   - Collections
   - Parallel and Concurrent Programming
   - Factoring Out Some Repeating Patterns
“Easy” Parallelism with Functional Programming

Higher-order functions are easier to handle.

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

map (\( \lambda x \rightarrow x + 2 \)) vect

easier to parallelise
```
“Easy” Parallelism with Functional Programming

Higher-order functions are easier to handle.

\[
\text{for}(i = 0; i < n; i++) \\
\text{vect}[i] = \text{vect}[i] + 2;
\]

\[
\text{map } (\lambda x \rightarrow x + 2) \text{ vect}
\]

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 $\rightarrow$ independent subproblems
2. solve subproblems independently
   - no shared resources

multiple threads share resources
   - synchronisation

Types allow static differentiation between parallel threads and concurrent threads.
Parallelism vs. Concurrency. Statically.

1. problem $\rightarrow$ independent subproblems
2. solve subproblems independently
   - no shared resources

multiple threads share resources
   - synchronisation

Types allow static differentiation between parallel threads and concurrent threads.
   - monads
Outline

1. Part 1
   - Calculus
   - Formal Languages
   - Set Theory

2. Part 2
   - Collections
   - Parallel and Concurrent Programming
   - Factoring Out Some Repeating Patterns
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

\[
\begin{align*}
\text{personByName} & : \ String \to \ Maybe \ Person \\
\text{carByPerson} & : \ Person \to \ Maybe \ Car \\
\text{model} & : \ Car \to \ Maybe \ String
\end{align*}
\]

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

\[
\text{case} \ \text{personByName} \ "John" \ \text{of} \\
\hspace{1cm} \text{Nothing} \to \ \text{Nothing} \\
\hspace{1cm} \text{Just} \ \text{john} \to
\]
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

\[
\text{personByName} :: \text{String} \rightarrow \text{Maybe Person} \\
\text{carByPerson} :: \text{Person} \rightarrow \text{Maybe Car} \\
\text{model} :: \text{Car} \rightarrow \text{Maybe String}
\]

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

\[
\text{case personByName "John" of} \\
\quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \text{Just john} \rightarrow \\
\qquad \text{case carByPerson john of} \\
\qquad \quad \text{Nothing} \rightarrow \text{Nothing} \\
\qquad \quad \text{Just johnsCar} \rightarrow \text{model johnsCar}
\]

Imagine what happens if one has longer chains.

https://openclipart.org/
Factoring out Patterns \( \text{Monads! \o/} \)

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