Simple Functional Data

COS 326

Andrew Appel

Princeton University

slides copyright 2022 David Walker and Andrew Appel
permission granted to reuse these slides for non-commercial educational purposes
What is the single most important mathematical concept ever developed in human history?
What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable
What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

(runner up: natural numbers/induction)
Why is the mathematical variable so important?

The mathematician says:

“Let x be some integer, we define a polynomial over x ...”
Why is the mathematical variable so important?

The mathematician says:

“Let \( x \) be some integer, we define a polynomial over \( x \) ...”

What is going on here? The mathematician has separated a definition (of \( x \)) from its use (in the polynomial).

This is the most primitive kind of abstraction (\( x \) is some integer).

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exist.

It allows reuse of ideas, theorems ... functions and programs!
OCAMLR BASICS:
LET DECLARATIONS
Abstraction & Abbreviation

In OCaml, the most basic technique for factoring your code is to use let expressions.

Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]

We write this one:

```ocaml
let x = 2 + 3 in
x * x
```
A Few More Let Expressions

let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs

let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
A Technical Note: The Structure of a .ml File

Every .ml file is a sequence of *declarations*

These “declarations” are a little different than “expressions”
A Technical Note: The Structure of a .ml File

Bar.ml contains two *let declarations*

Let declarations do not end with “in”

Let declarations have the form:

```plaintext
let <var> = <expression>
```
A Technical Note: The Structure of a .ml File

Because let declarations have this form:

```ml
let <var> = <expression>
```

they contain expressions

... including “let expressions” which have the form:

```ml
let <var> = <expression> in <expression>
```

Baz.ml

```ml
let x =
    let z = 22 in
    z + z

let y =
    if x < 17 then
        let w = x + 1 in
        2 * w
    else
        26
```
OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```ocaml
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for `x`, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*. 
Once *bound* to a value, a variable is never modified or changed.

```ocaml
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for `x`, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*. 
OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```ocaml
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for `x`, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*. 
OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```ocaml
let x = 3

let add_three (y:int) : int = y + x
```

It does not matter what I write next. *add_three* will always add 3!
OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```ocaml
let x = 3
let add_three (y:int) : int = y + x

let x = 4
let add_four (y:int) : int = y + x
```

a distinct variable that "happens to be spelled the same"
OCaml Variables are Immutable

A use of a variable always refers to it’s closest (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a *statically scoped* (or *lexically scoped*) language.

```
let x = 3
let add_three (y:int) : int = y + x

let x = 4
let add_four (y:int) : int = y + x

let add_seven (y:int) : int = add_three (add_four y)
```

we can use add_three without worrying about the second definition of x
OCaml Variables are Immutable

Since the two variables (both happened to be named x) are actually different, unconnected things, we can rename them. This is known as *alpha-conversion*.

```
let x = 3
let add_three (y:int) : int = y + x

let x = 4
let add_four (y:int) : int = y + x

let add_seven (y:int) : int = add_three (add_four y)
```

you can rename x to zzz by replacing the definition and all its uses with the new name
OCaml Variables are Immutable

Since the two variables (both happened to be named x) are actually different, unconnected things, we can rename them. This is known as *alpha-conversion*.

```ocaml
let x = 3

let add_three (y:int) : int = y + x

let zzz = 4

let add_four (y:int) : int = y + zzz

let add_seven (y:int) : int =
  add_three (add_four y)
```

you can rename x to zzz by replacing the definition and all its uses with the new name.
How does OCaml execute a let expression?

In a nutshell:
• execute `<expression1>`, until you get a value v1
• substitute that value v1 for x in `<expression2>`
• execute `<expression2>`, until you get a value v2
• the result of the whole execution is v2
How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```
How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

--> 

```
let x = 3 in x * x
```

--> 

```
3 * 3
```

substitute 3 for x
How does OCaml execute a let expression?

\[
\text{let } x = 2 + 1 \text{ in } x \times x
\]

\[
\Rightarrow
\text{let } x = 3 \text{ in } x \times x
\]

\[
\Rightarrow
3 \times 3
\]

\[
9
\]

substitute 3 for x
How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

-->

```
3 * 3
```

-->

```
9
```

Note: I write $e_1 \rightarrow e_2$ when $e_1$ evaluates to $e_2$ in one step.
I defined the language in terms of itself:
By reduction of one OCaml expression to another

I’m trying to train you to think at a high level of abstraction.

I didn’t have to mention low-level abstractions like assembly code or registers or memory layout to tell you how OCaml works.
Another Example

let x = 2 in
let y = x + x in
y * x
Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute 2 for x

```
let y = 2 + 2 in
y * 2
```

-->
Another Example

let x = 2 in
let y = x + x in
y * x

substitute 2 for x

--> let y = 2 + 2 in
     y * 2

--> let y = 4     in
     y * 2
Another Example

```
let x = 2 in
let y = x + x in
y * x
```

---

```
let y = 2 + 2 in
y * 2
```

---

```
let y = 4 in
y * 2
```

---

```
4 * 2
```
Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute 2 for x

```
let y = 2 + 2 in
y * 2
```

substitute 4 for y

```
let y = 4 in
y * 2
```

```
4 * 2
```

```
8
```

Moral: Let operates by **substituting** computed values for variables.
OCAML BASICS:
TYPE CHECKING AGAIN
Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in
x
```

overall expression takes on the type of e2
x granted type of e₁ for use in e₂

```
let x = e₁ in
e₂
```

overall expression takes on the type of e₂

x has type int for use inside the let body

```
let x = 3 + 4 in
string_of_int x
```

overall expression has type string
Let Expressions Really Are Expressions

2 + 3

an expression
Let Expressions Really Are Expressions

2 + 3

let x = 2 + 3 in
x + x

an expression
Let Expressions Really Are Expressions

\[ 2 + 3 \]

\[ \text{let } x = 2 + 3 \text{ in } x + x \]

\[ \text{let } y = 2 + 3 \text{ in } y + 5 \text{ in } 1 + x \]

Let expressions can appear anywhere other expressions can appear. They can be *nested*.
Which of (a) or (b) type check? Explain why.

On a piece of paper (or in your favorite editor), show the step-by-step evaluation of the example that type checks.

Critique the *programming style* used in these examples.
OCAML BASICS:
FUNCTIONS
let add_one (x:int) : int = 1 + x
let add_one (x:int) : int = 1 + x

---

Note: recursive functions with begin with "let rec"
Defining functions

Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x
let add_two (x:int) : int = add_one (add_one x)
```
Defining functions

Nonrecursive functions:

let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)

With a local definition:

let add_two' (x:int) : int =
let add_one x = 1 + x in
add_one (add_one x)
Types for Functions

Some functions:

```ocaml
let add_one (x:int) : int = 1 + x
let add_two (x:int) : int = add_one (add_one x)
let add (x:int) (y:int) : int = x + y
```

Types for functions:

```ocaml
add_one : int -> int
add_two : int -> int
add : int -> int -> int
```
Rule for type-checking functions

General Rule:

If a function $f : T_1 \rightarrow T_2$ and an argument $e : T_1$ then $f \, e : T_2$

Example:

add_one : int -> int
3 + 4 : int
add_one (3 + 4) : int
Rule for type-checking functions

Recall the type of add:

**Definition:**

let add (x:int) (y:int) : int =
    x + y

**Type:**

add : int -> int -> int
Rule for type-checking functions

Recall the type of add:

**Definition:**

```ocaml
let add (x:int) (y:int) : int =
    x + y
```

**Type:**

```ocaml
add : int -> int -> int
```

**Same as:**

```ocaml
add : int -> (int -> int)
```
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f e : T_2 \)

Example:

\[
\begin{align*}
\text{add} & : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 & : \text{int} \\
\text{add} \ (3 + 4) & : ???
\end{align*}
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\[
\text{add} : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \\
3 + 4 : \text{int} \\
\text{add} \ (3 + 4) :
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \, e : T_2 \)

Example:

\[
\begin{align*}
\text{add} : \text{int} & \rightarrow (\text{int} \rightarrow \text{int}) \\
3 + 4 & : \text{int} \\
\text{add} \, (3 + 4) & : \text{int} \rightarrow \text{int}
\end{align*}
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f \, e : T_2 \)

Example:

\[
\text{add} : \, \text{int} \to \text{int} \to \text{int} \\
3 + 4 : \text{int} \\
\text{add} \, (3 + 4) : \text{int} \to \text{int} \\
(\text{add} \, (3 + 4)) \, 7 : \text{int}
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f e : T_2 \)

Example:

\[
\begin{align*}
\text{add} &: \text{int} \to \text{int} \to \text{int} \\
3 + 4 &: \text{int} \\
\text{add} (3 + 4) &: \text{int} \to \text{int} \\
\text{add} (3 + 4) 7 &: \text{int}
\end{align*}
\]
One key thing to remember

• If you have a function f with a type like this:

\[ A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \]

• Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

\[
\begin{align*}
  f\ a1 : &\ B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a1 : A) \\
  f\ a1\ a2 : &\ C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a2 : B) \\
  f\ a1\ a2\ a3 : &\ D \rightarrow E \rightarrow F \quad \text{(if } a3 : C) \\
  f\ a1\ a2\ a3\ a4\ a5 : &\ F \quad \text{(if } a4 : D \text{ and } a5 : E)
\end{align*}
\]
TYPE ERRORS
Type errors for if statements can be confusing sometimes. Recall:

```ocaml
let rec concatn s n = 
  if n <= 0 then ... 
  else s ^ (concatn s (n-1))
```
Type errors for if statements can be confusing sometimes. Recall:

```ocaml
let rec concatn s n =  
  if n <= 0 then  
    ...  
  else  
    s ^ (concatn s (n-1))
```

ocaml might point to (concatn s (n-1)) and says:

Error: This expression has type int but an expression was expected of type string
Type errors for if statements can be confusing sometimes. Recall:

```ocaml
let rec concatn s n =
  if n <= 0 then ...
  else
    s ^ (concatn s (n-1))
```

ocaml might say:

```
Error: This expression has type int but an expression was expected of type string
```

or ocaml might point to the expression `(s ^ (concatn ...))` and say:

```
Error: This expression has type string but an expression was expected of type int
```
Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```ocaml
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

Error: This expression has type int but an expression was expected of type string

Error: This expression has type string but an expression was expected of type int
Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

they don't agree!

Error: This expression has type int but an expression was expected of type string

Error: This expression has type string but an expression was expected of type int
Type errors for if statements can be confusing sometimes. Example. We create a string from s, concatenating it n times:

```ml
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

The type checker points to some place where there is disagreement.

Moral: Sometimes you need to look in an earlier branch for the error even though the type checker points to a later branch. The type checker doesn't know what the user wants.
A Tactic: Add Typing Annotations

```ocaml
let rec concatn (s:string) (n:int) : string =
    if n <= 0 then
        0
    else
        s ^ (concatn s (n-1))
```

Error: This expression has type int but an expression was expected of type string
Exercise

Given the following code:

```ocaml
let munge b x =
  if not b then
    string_of_int x
  else
    "hello"

let y = 17
```

What are the types of the following expressions?  (And what must the types of f and g be?)

```ocaml
munge : ??
munge (y > 17) : ??
munge true (f (munge false 3)) : ??
munge true (g munge) : ??
```
DATA STRUCTURES:
THE TUPLE

* it is really our second complex data structure since functions are data structures too!
A tuple is a fixed, finite, ordered collection of values

Some examples with their types:

(1, 2) : int * int
("hello", 7 + 3, true) : string * int * bool
('a', ("hello", "goodbye")) : char * (string * string)
To use a tuple, we extract its components

General case:

let (id1, id2, ..., idn) = e1 in e2

An example:

let (x,y) = (2,4) in x + x + y
To use a tuple, we extract its components

General case:

\[
\text{let } (id_1, id_2, \ldots, id_n) = e_1 \text{ in } e_2
\]

An example:

\[
\text{let } (x, y) = (2, 4) \text{ in } x + x + y \\
\rightarrow 2 + 2 + 4
\]

substitute!
To use a tuple, we extract its components

General case:

\[
\text{let } (id_1, id_2, \ldots, id_n) = e_1 \text{ in } e_2
\]

An example:

\[
\text{let } (x, y) = (2, 4) \text{ in } x + x + y
\]

\[
\begin{align*}
&\rightarrow 2 + 2 + 4 \\
&\rightarrow 8
\end{align*}
\]
Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2
Rules for Typing Tuples

\[
\text{if } e_1 : t_1 \text{ and } e_2 : t_2 \\
\text{then } (e_1, e_2) : t_1 * t_2
\]

if \( e_1 : t_1 \ast t_2 \) then
\( x_1 : t_1 \) and \( x_2 : t_2 \)
inside the expression \( e_2 \)

let \((x_1, x_2) = e_1 \) in
\( e_2 \)

overall expression takes on the type of \( e_2 \)
Problem:
- A point is represented as a pair of floating point values.
- Write a function that takes in two points as arguments and returns the distance between them as a floating point number.
Steps to writing functions over typed data:

1. **Write down** the function and argument names
2. **Write down** argument and result **types**
3. **Write down** some examples (in a comment)
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   - the argument types suggests how to do it
5. Build new output values
   - the result type suggests how you do it
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   • the argument types suggest how to do it
5. Build new output values
   • the result type suggests how you do it
6. Clean up by identifying repeated patterns
   • define and reuse helper functions
   • your code should be elegant and easy to read
Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names
2. **Write down** argument and result **types**
3. Write down some examples (in a comment)
4. **Deconstruct** input data structures
   - *the argument types suggest how to do it*
5. **Build** new output values
   - *the result type suggests how you do it*
6. Clean up by identifying repeated patterns
   - define and reuse helper functions
   - your code should be elegant and easy to read

*Types help structure your thinking about how to write programs.*
Distance between two points

type point = float * float

(a type abbreviation)

\[(x_1, y_1)\] 
\[(x_2, y_2)\]
type point = float * float

let distance (p1:point) (p2:point) : float =

write down function name
argument names and types
**Distance between two points**

**Type**

\[
\text{type point = float * float}
\]

**Examples**

\[
\begin{align*}
\text{(* distance (0.0,0.0) (0.0,1.0) == 1.0)} \\
\text{(* distance (0.0,0.0) (1.0,1.0) == sqrt(1.0 + 1.0)} \\
\text{(* from the picture:)} \\
\text{(* distance (x1,y1) (x2,y2) == sqrt(a^2 + b^2)} \\
\text{*)}
\end{align*}
\]

**Function**

\[
\text{let distance (p1:point) (p2:point) : float =}
\]

\[
\begin{align*}
\text{(x1, y1)} \\
\text{(x2, y2)} \\
\text{a} \\
\text{b} \\
\text{c}
\end{align*}
\]
Distance between two points

type point = float * float

let distance (p1:point) (p2:point) : float =

  let (x1, y1) = p1 in
  let (x2, y2) = p2 in
  ...

  deconstruct function inputs
Distance between two points

```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt ((x2 -. x1) *. (x2 -. x1) +. (y2 -. y1) *. (y2 -. y1))
```

Notice operators on floats have a "." in them.

Compute function results.
Distance between two points

type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1)) +. 
    square (y2 -. y1))

define helper functions to avoid repeated code
type point = float * float

let distance (x1, y1) (x2, y2) =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))

use tuple patterns in function arguments if you’d like
type point = float * float

let distance ((x1, y1):point) ((x2, y2):point): float =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))

let pt1 = (2.0,3.0)
let pt2 = (0.0,1.0)
let dist12 = distance pt1 pt2
MORE TUPLES
Here's a tuple with 2 fields:

\[(4.0, 5.0) : \text{float} \times \text{float}\]
Tuples

Here's a tuple with 2 fields:

(4.0, 5.0) : float * float

Here's a tuple with 3 fields:

(4.0, 5, "hello") : float * int * string
Tuples

Here's a tuple with 2 fields:

(4.0, 5.0) : float * float

Here's a tuple with 3 fields:

(4.0, 5, "hello") : float * int * string

Here's a tuple with 4 fields:

(4.0, 5, "hello", 55) : float * int * string * int
Tuples

Here's a tuple with 2 fields:

\[(4.0, 5.0) : \text{float} \times \text{float}\]

Here's a tuple with 3 fields:

\[(4.0, 5, "hello") : \text{float} \times \text{int} \times \text{string}\]

Here's a tuple with 4 fields:

\[(4.0, 5, "hello", 55) : \text{float} \times \text{int} \times \text{string} \times \text{int}\]

Here's a tuple with 0 fields:

\[() : \text{unit}\]
SUMMARY:
BASIC FUNCTIONAL PROGRAMMING
Steps to writing functions over typed data:
1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. **Deconstruct** input data structures
5. **Build** new output values
6. Clean up by identifying repeated patterns

For tuple types:
- when the **input** has type $t1 \times t2$
  - use `let (x,y) = ...` to deconstruct
- when the **output** has type $t1 \times t2$
  - use `(e1, e2)` to construct

We will see this paradigm repeat itself over and over
Records

Records are a lot like tuples. It’s just that they have named fields.

Having named fields (records rather than tuples) often makes it easier to understand a program, especially when there are more than just 2 or 3 fields in a structure.
Records

Records are a lot like tuples. It’s just that they have named fields.

Having named fields (records rather than tuples) often makes it easier to understand a program, especially when there are more than just 2 or 3 fields in a structure.

An example:

```plaintext
type name = {first:string; last:string;}
let my_name = {first="David"; last="Walker"];}
let to_string (n:name) = n.last ^ ", " ^ n.first
```
Records

Records are a lot like tuples. It’s just that they have named fields.

Having named fields (records rather than tuples) often makes it easier to understand a program, especially when there are more than just 2 or 3 fields in a structure.

An example:

```ocaml
type name = {first:string; last:string;}  
let my_name = {first="David"; last="Walker";}  
let to_string (n:name) = n.last ^ "", " ^ n.first 
```

Note: Records come with several other useful features, like functional updates via “with expressions.”

See *Real World OCaml* for more info.
WRAP-UP
Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. **Deconstruct** input data structures
5. **Build** new output values
6. Clean up by identifying repeated patterns

For tuple types:

- when the **input** has type $t_1 \times t_2$
  - use `let (x,y) = ...` to deconstruct
- when the **output** has type $t_1 \times t_2$
  - use `(e1, e2)` to construct

We will see this paradigm repeat itself over and over
What error do you get when you try to compile this file? (Type it in.) Why?

```plaintext
type item = {
    number: int;
    name: string;
}

type contact = {
    name: string*string; (* first and last name *)
    phone: item;
}

let get_name x = x.name

let myphone = {number=122; name="iphone";}

let _ = print_endline (get_name myphone)
```
WHERE DID TYPE SYSTEMS COME FROM?
Origins of Type Theory

Georg Cantor

Origins of Type Theory

Georg Cantor

Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. 1874

(On a Property of the System of all the Real Algebraic Numbers)

“Considered the first purely theoretical paper on set theory.” *

He noticed that Cantor’s set theory allows the definition of this set $S$:

$$\{ A \mid A \text{ is a set and } A \notin A \}$$

Bertrand Russell
He noticed that Cantor’s set theory allows the definition of this set $S$:

$$\{ A \mid A \text{ is a set and } A \notin A \}$$

If we assume $S$ is not in the set $S$, then by definition, it must belong to that set.

If we assume $S$ is in the set $S$, then it contradicts the definition of $S$.

Russell’s paradox
He noticed that Cantor’s set theory allows the definition of this set $S$:

\[ \{ A \mid A \text{ is a set and } A \notin A \} \]

Russell’s solution:

Each set has a distinct type: type 1, 2, 3, 4, 5, ...

A set of type $i+1$ can only have elements of type $i$ so it can’t include itself.
Aside

Ernst Zermelo
Abraham Fraenkel

Developers of Zermelo-Fraenkel set theory (1921).
An alternative solution to Russell’s paradox.
Origins of Type Theory

Alonzo Church, 1903-1995
Princeton Professor, 1929-1967

Developed the lambda calculus
(ancestor of ML / OCaml)

and "The simple theory of types"
(ancestor of ML's type system)