Simple Data

COS 326
David Walker
Princeton University

slides copyright 2017 David Walker
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!
OCAML BASICS:
LET DECLARATIONS
Abstraction

• Good programmers identify repeated patterns in their code and factor out the repetition into meaningful components

• In O’Caml, the most basic technique for factoring your code is to use let expressions

• Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]
Abstraction & Abbreviation

- Good programmers identify repeated patterns in their code and factor out the repetition into meaning components.
- In O’Caml, 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
```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
A Few More Let Expressions

let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed

let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs
Abstraction & Abbreviation

- Two kinds of let:

  let ... in ... is an expression that can appear inside any other expression

  The scope of x does not extend outside the enclosing “in”

  let x = 2 + 3
  x + x

  let y = x + 17 / x

  let ... without “in” is a top-level declaration

  Variables x and y may be exported; used by other modules

  (Don’t need ;; if another let comes next; do need it if the next top-level declaration is an expression)
Binding Variables to Values

• Each OCaml variable is *bound* to 1 value
• *The value to which a variable is bound to never changes!*

```ocaml
let x = 3

let add_three (y:int) : int = y + x
```
Binding Variables to Values

- Each OCaml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes!*

```
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!*
Binding Variables to Values

- Each OCaml variable is bound to 1 value
- *The value a variable is bound to never changes!*

```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"
Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them.

Let's rename x to zzz if you want to, replacing its uses.

```plaintext
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)
```
Binding Variables to Values

- Each OCaml variable is bound to 1 value
- OCaml is a **statically scoped** (or **lexically scoped**) language

```ocaml
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
How do let expressions operate?

```
let x = 2 + 1 in x * x
```
How do let expressions operate?

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

--> 

```plaintext
let x = 3 in x * x
```
How do let expressions operate?

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

-->

let x = 3 in x * x

-->

3 * 3
```

substitute 3 for x
How do let expressions operate?

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

$\rightarrow$

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

$\rightarrow$

```latex
3 * 3
```

$\rightarrow$

```latex
9
```

substitute 3 for x
How do let expressions operate?

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

\[=\]

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

\[=\]

\[
3 \times 3
\]

\[=\]

\[
9
\]

Note: I write \( e_1 \rightarrow e_2 \) when \( e_1 \) evaluates to \( e_2 \) in one step.
Did you see what I did there?
Did you see what I did there?

I defined the language in terms of itself:

```plaintext
let x = 2 in x + 3  --&gt;  2 + 3
```

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*
Another Example

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

\[
\begin{align*}
\text{let } x &= 2 \text{ in } \\
\text{let } y &= x + x \text{ in } \\
y &= x \\
\end{align*}
\]

\[
\begin{align*}
\text{let } y &= 2 + 2 \text{ in } \\
y &= 4 \\
\end{align*}
\]

substitute 2 for x
Another Example

```plaintext
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
```

substitute 2 for x

substitute 4 for y
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

```
4 * 2
```

**Moral:** Let operates by *substituting* computed values for variables

```
8
```
What would happen in an imperative language?

C program:

```c
x = 2;
x += x;
return x*2;
```

substitute 2 for \( x \)

--> 

```c
x += 2 ???
return x*2;
```

substituting computed values for variables

This principle works in functional languages, not so well in imperative languages
OCAML BASICS: TYPE CHECKING AGAIN
Back to Let Expressions ... Typing

\[ \text{x granted type of } e_1 \text{ for use in } e_2 \]

\[
\text{let } x = e_1 \text{ in } e_2
\]

overall expression takes on the type of \( e_2 \)
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
OCAML BASICS: FUNCTIONS
let add_one (x:int) : int = 1 + x
let keyword

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

function name

argument name

argument type

result type

expression that computes value produced by function

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

- Nonrecursive functions:

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

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

definition of add_one must come before use
Defining functions

• Nonrecursive functions:

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

• With a local definition:

```ocaml
let add_two' (x:int) : int =
  let add_one x = 1 + x in
  add_one (add_one x)
```

I left off the types. O'Caml figures them out.

Good style: types on top-level definitions. 
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 \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f\ e : T_2 \)

Example:

\[
\begin{align*}
\text{add\_one} & : \text{int} \to \text{int} \\
3 + 4 & : \text{int} \\
\text{add\_one} (3 + 4) & : \text{int}
\end{align*}
\]
• 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:

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

Type:

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

Same as:

```
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 \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\[
\text{add : int} \to (\text{int} \to \text{int})
\]

\[
3 + 4 : \text{int}
\]

\[
\text{add (3 + 4)} :
\]
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:

\[
\begin{align*}
\text{add} &: \text{int} \to (\text{int} \to \text{int}) \\
3 + 4 &: \text{int} \\
\text{add} \, (3 + 4) &: \text{int} \to \text{int}
\end{align*}
\]
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:

\[
\begin{align*}
\text{add} : & \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 : & \text{int} \\
\text{add} (3 + 4) : & \text{int} \rightarrow \text{int} \\
(\text{add} (3 + 4)) \ 7 : & \text{int}
\end{align*}
\]
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:

```plaintext
add : int -> int -> int
3 + 4 : int
add (3 + 4) : int -> int
add (3 + 4) 7 : int
```

A -> B -> C

same as:

A -> (B -> C)
Rule for type-checking functions

Example:

let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;

munge (y > 17) : ??
munge true (f (munge false 3)) : ??
  f : ??
munge true (g munge) : ??
  g : ??
Example:

```ocaml
let munge (b:bool) (x:int) : ?? = 
  if not b then 
    string_of_int x 
  else 
    "hello"
;;

let y = 17;;

munge (y > 17) : ??

munge true (f (munge false 3)) : ?? 
  f : string -> int

munge true (g munge) : ??
  g : (bool -> int -> string) -> int
```
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

\[
f \ a1 : B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if a1 : A)}
\]

\[
f \ a1 \ a2 : C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if a2 : B)}
\]

\[
f \ a1 \ a2 \ a3 : D \rightarrow E \rightarrow F \quad \text{(if a3 : C)}
\]

\[
f \ a1 \ a2 \ a3 \ a4 \ a5 : F \quad \text{(if a4 : D and a5 : E)}
\]
OUR FIRST* COMPLEX DATA STRUCTURE!
THE TUPLE

* it is really our second complex data structure since functions are data structures too!
Tuples

• 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)
Tuples

• To use a tuple, we extract its components

• General case:

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

• An example:

```plaintext
let (x, y) = (2, 4) in x + x + y
```
• To use a tuple, we extract its components
• General case:

\[
\text{let } (id1, id2, \ldots, idn) = e1 \text{ in } e2
\]

• An example:

\[
\text{let } (x, y) = (2, 4) \text{ in } x + x + y \\
\rightarrow 2 + 2 + 4
\]
• To use a tuple, we extract its components
• General case:

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

• An example:

```plaintext
let (x, y) = (2, 4) in x + x + y
--> 2 + 2 + 4
--> 8
```
if $e_1 : t_1$ and $e_2 : t_2$
then $(e_1, e_2) : t_1 * t_2$
**Rules for Typing Tuples**

If \( e_1 : t_1 \) and \( e_2 : t_2 \) then \( (e_1, e_2) : t_1 \times t_2 \)

If \( e_1 : t_1 \times t_2 \) 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 \( e_2 \)
Distance between two points

\[ c^2 = a^2 + b^2 \]

**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.
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)
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*
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*
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 suggests 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

a type abbreviation

type point = float * float
Distance between two points

```ml
type point = float * float

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

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

```ocaml
type point = float * float

(* distance (0.0,0.0) (0.0,1.0) == 1.0
 * distance (0.0,0.0) (1.0,1.0) == sqrt(1.0 + 1.0)
 * from the picture:
 * distance (x1,y1) (x2,y2) == sqrt(a^2 + b^2)
 *)

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

examples
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))

compute function results

notice operators on floats have a "." in them
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 (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
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:

  \((4.0, 5.0) : \text{float} \ast \text{float}\)

• Here's a tuple with 3 fields:

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

• Here's a tuple with 0 fields:

() : unit
• **Unit** is the tuple with zero fields!

\[
() : \text{unit}
\]

• the unit value is written with a pair of parens
• there are no other values with this type!
**Unit**

- **Unit** is the tuple with zero fields!

\[
() : \text{unit}
\]

- the unit value is written with a pair of parens
- there are no other values with this type!

- Why is the unit type and value useful?
- Every expression has a type:

\[
\text{(print\_string "hello world\n") : ???}
\]
Unit

• Unit is the tuple with zero fields!

  \((()) : \text{unit}\)

  • the unit value is written with an pair of parens
  • there are no other values with this type!

• Why is the unit type and value useful?
• Every expression has a type:

  \((\text{print\_string } "\text{hello world}\n") : \text{unit}\)

• Expressions executed for their \textit{effect} return the unit value
SUMMARY:
BASIC FUNCTIONAL PROGRAMMING
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