Simple Functional Data

COS 326

Andrew Appel

Princeton University
TYPE ERRORS
Type errors for if statements can be confusing sometimes. Recall:

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

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))
```

ocamlbuild 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))
```

ocamlbuild says:

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

merlin inside emacs points to the error above and gives a second error:

```
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))
```

ocamlbuild says:

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

merlin inside emacs points to the error above and gives a second error:

```
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
    0
  else
    s ^ (concatn s (n-1))
```

they don't agree!

ocamlbuild says:

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

merlin inside emacs points to the error above and gives a second error:

```
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.
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
ONWARD
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:

\[
\text{let } x = 2 + 3 \text{ in } x \times x
\]
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
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
```
Abstraction & Abbreviation

Two “kinds” of let:

- **let ... in ...** is an *expression* that can appear inside any other *expression*
- The scope of x (ie: the places x may be used) does not extend outside the enclosing “in”

- **let ...** without “in” is a top-level *declaration*
- Variables x and y may be exported; used by other modules
- You can only omit the “in” in a top-level declaration
Binding Variables to Values

During execution, we say an OCaml variable is \textit{bound} to a value. \textit{The value to which a variable is bound to never changes!}

\begin{verbatim}
let x = 3
let add_three (y:int) : int = y + x
\end{verbatim}
Binding Variables to Values

During execution, we say an OCaml variable is *bound* to a 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!*
During execution, we say an OCaml variable is *bound* to a value. *The value to which a variable is bound to never changes!*
Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

```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)
```

rename x to zzz if you want to, replacing its uses
A use of a variable always refers to its closest (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a \textit{statically scoped} (or \textit{lexically scoped}) language.

\begin{align*}
\text{let } x &= 3 \\
\text{let } \text{add}_\text{three} \ (y: \text{int}) : \text{int} &= y + x \\
\text{let } x &= 4 \\
\text{let } \text{add}_\text{four} \ (y: \text{int}) : \text{int} &= y + x \\
\text{let } \text{add}_\text{seven} \ (y: \text{int}) : \text{int} &= \text{add}_\text{three} \ (\text{add}_\text{four} \ y)
\end{align*}
How does OCaml execute a let expression?

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

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

-->

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

--> substitute 3 for x

```ocaml
3 * 3
```

-->

```ocaml
9
```
How does OCaml execute a let expression?

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

\[\rightarrow\]

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

\[\rightarrow\]

\[
3 \times 3
\]

\[\rightarrow\]

\[
9
\]

Note: I write \(e_1 \rightarrow e_2\) when \(e_1\) evaluates to \(e_2\) in one step.
let x = 2 in x + 3 \rightarrow 2 + 3

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

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

\text{substitute} 2 \text{ for } x

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

\text{let } y &= 2 + 2 \text{ in} \\
y &= 4

\text{let } y &= 4 \text{ in} \\
y &= 4
Another Example

```plaintext
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
```

substitute 4 for y

```
4 * 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

-->

8

Moral: Let operates by substituting computed values for variables
OCAML BASICS:
TYPE CHECKING AGAIN
x granted type of e1 for use in e2

let x = e1 in

overall expression takes on the type of e2
Back to Let Expressions ... Typing

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

Defining functions

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
let 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

local function definition hidden from clients
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:

- `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:

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

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

Type:

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

Same as:

```plaintext
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:

\[
\text{add} : \text{int} \rightarrow \text{int} \rightarrow \text{int}
\]

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

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

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

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

\[
\text{add : int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 : \text{int} \\
\text{add}(3 + 4) : \text{int} \rightarrow \text{int} \\
\text{add}(3 + 4) 7 : \text{int}
\]

\( A \rightarrow B \rightarrow C \)

same as:

\( A \rightarrow (B \rightarrow C) \)

extra parens not necessary
Example:

```haskell
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 : ??
```
Rule for type-checking functions

Example:

```plaintext
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

\[
\begin{align*}
  f \text{ a1 : } B & \rightarrow C \rightarrow D \rightarrow E \rightarrow F & \text{(if a1 : A)} \\
  f \text{ a1 a2 : } C & \rightarrow D \rightarrow E \rightarrow F & \text{(if a2 : B)} \\
  f \text{ a1 a2 a3 : } D & \rightarrow E \rightarrow F & \text{(if a3 : C)} \\
  f \text{ a1 a2 a3 a4 a5 : } F & & \text{(if a4 : D and a5 : E)}
\end{align*}
\]
OUR FIRST* COMPLEX DATA STRUCTURE!
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:

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

An example:

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

--> 2 + 2 + 4
```

substitute!
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
--> 2 + 2 + 4
--> 8
```
Rules for Typing Tuples

if \texttt{e1 : t1} and \texttt{e2 : t2}
then \texttt{(e1, e2) : t1 \ast t2}
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.
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)
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
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

type point = float * float

(a, b)  (x1, y1)
(c, d)  (x2, y2)
Distance between two points

type point = float * float

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

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

\[
\begin{aligned}
\text{type point} &= \text{float } \times \text{float} \\
(* \text{ distance } (0.0,0.0) (0.0,1.0) == 1.0 \\
* \text{ distance } (0.0,0.0) (1.0,1.0) == \sqrt{1.0 + 1.0} *) \\
* \text{ from the picture: } \\
* \text{ distance } (x_1,y_1) (x_2,y_2) == \sqrt{a^2 + b^2} *) \\
\end{aligned}
\]

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

```ocaml
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.
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
Here's a tuple with 2 fields:

(4.0, 5.0) : float * float
Here's a tuple with 2 fields:

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

Here's a tuple with 3 fields:

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

Origins of Type Theory

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 \}
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.
Fast Forward to the 70s

In 1978, developed ML and coined the phrase “well-typed programs don’t go wrong” to describe a key property of type-safe languages.

Robin Milner
Well-typed Programs Don’t Go Wrong

Some ML programs do not have a well-defined semantics:

```
“hello” + 3
```

Such programs do not type check.
Some ML programs do not have a well-defined semantics:

> let x = "hello" in
> let y = 3 in
> x + y

Such programs do not type check.

Moreover, when we execute a well-typed program, *we are guaranteed to never, ever run into such a program during execution.*

> "hello" + 3

> let x = "hello" in
> let y = 3 in
> x + y
Well-typed Programs Don’t Go Wrong

Some ML programs do not have a well-defined semantics:

```
let x = "hello" in
let y = 3 in
x + y
```

"hello" + 3

Such programs do not type check.

Moreover, when we execute a well-typed program, we are guaranteed to never, ever run into such a program during execution.

```
let x = "hello" in
let y = 3 in
x + y
```

-->*  "hello" + 3

well-typed programs don’t reduce to programs like “hello” + 3, which go wrong
Well-type programs don’t go wrong

What about this expression:

$$3 \div 0$$
Well-type programs don’t go wrong

What about this expression:

```
3 / 0
```

It type checks. When executed, ML will supply this message:

```
Exception: Division_by_zero.
```

Did the expression “go wrong”?
Did it violate our credo “well-typed expressions don’t go wrong?”
Well-type programs don’t go wrong

What about this expression:

3 / 0

It type checks. When executed, ML will supply this message:

Exception: Division_by_zero.

Did the expression “go wrong”? Did it violate our credo “well-typed expressions don’t go wrong?”

No and No. Exceptions are a well-defined result of a computation. ie: you can look up what happens to 3 / 0 in the OCaml manual.
What’s the difference between raising an exception and “going wrong”? Why distinguish between these things? Does one have to treat “hello” + 3 as “going wrong”? Why does OCaml make such choices? Is it reasonable for other languages to choose differently?
Type Soundness

“Well typed programs do not go wrong”

Programming languages with this property have sound type systems. They are called safe languages.

Safe languages are generally immune to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc. (but not immune to all bugs!)

Safe languages: ML, Java, Python, ...

Unsafe languages: C, C++, Pascal
Well typed programs do not go wrong

Robin Milner

Turing Award, 1991

“For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.”

“Well typed programs do not go wrong”

Robin Milner, 1978