Tuples

Speaker: David Walker
COS 326
Princeton University
A tuple is a fixed, finite, ordered collection of values
A tuple is a fixed, finite, ordered collection of values

Some examples with their types:

\[(1, 2) : \text{int} \times \text{int}\]
\[\text{"hello", 7 + 3, true} : \text{string} \times \text{int} \times \text{bool}\]
\[\text{'a', ("hello", "goodbye")} : \text{char} \times (\text{string} \times \text{string})\]
To use a tuple, we extract its components

General case:

\[
\text{let } (\text{id}_1, \text{id}_2, \ldots, \text{id}_n) = \text{e}_1 \text{ in } \text{e}_2
\]
Tuples

To use a tuple, we extract its components

General case:

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

A concrete example:

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

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

```
let (x, y) = (2, 4) in x + x + y
```

```
2 + 2 + 4
```

substitute!
Evaluating Tuple Expressions

let \((x, y) = (2, 4)\) in \(x + x + y\)

\[
\begin{align*}
\text{let } (x, y) &= (2, 4) \text{ in } x + x + y \\
\rightarrow & \quad 2 + 2 + 4 \\
\rightarrow & \quad 4 + 4 \\
\rightarrow & \quad 8
\end{align*}
\]
Rules for Typing Tuples

if $e_1 : t_1$ and $e_2 : t_2$
then $(e_1, e_2) : t_1 \times t_2$
Rules for Typing Tuples

if $e_1 : t_1$ and $e_2 : t_2$
then $(e_1, e_2) : t_1 * t_2$

if $e_1 : t_1 * t_2$ 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 $e_2$
DEVELOPING PROGRAMS
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 suggests how to do it
5. Build new output values
   - the result type suggests how you do it
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
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 \textbf{types}
3. Write down some examples (in a comment)
4. \textbf{Deconstruct} input data structures
   \begin{itemize}
   \item the \textit{argument types} suggests how to do it
   \end{itemize}
5. \textbf{Build} new output values
   \begin{itemize}
   \item the \textit{result type} suggests how you do it
   \end{itemize}
6. Clean up by identifying repeated patterns
   \begin{itemize}
   \item define and reuse helper functions
   \item your code should be elegant and easy to read
   \end{itemize}

\textit{Types help structure your thinking about how to write programs.}
Distance between two points

Type abbreviation

(type point = float * float)
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

**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 =
Distances 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
  ...
```

deconstruct function inputs
Distance between two points

**Type Definition**

```plaintext
type point = float * float
```

**Function Definition**

```plaintext
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))
```

**Diagram**

- Points: (x1, y1) and (x2, y2)
- Distance formula: \[ \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]

**Notes**

- Operators on floats have a "." in them.
- The computed function results in the distance between the two points.
Distance between two points

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)
Distance between two points

```
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
Distance between two points

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 annotations can be included
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

Implement some tests
MORE TUPLES
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) : float * float

Here's a tuple with 3 fields:

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

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

Why is it useful to have a tuple with zero fields?
Unit

Why is it useful to have a tuple with zero fields?

• Every expression in OCaml returns *some value*

• We need a value to return when we call a function that doesn’t return any data …

• … but what good is a function that returns no data?
Why is it useful to have a tuple with zero fields?

• Every expression in OCaml returns *some value*

• We need a value to return when we call a function that doesn’t return any data …

• … but what good is a function that returns no data?

Some functions have *effects*, which do their work:

• Functions that print to the terminal:

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

Why is it useful to have a tuple with zero fields?
- Every expression in OCaml returns *some value*
- We need a value to return when we call a function that doesn’t return any data …
- … but what good is a function that returns no data?

Some functions have **effects**, which do their work:
- Functions that print to the terminal:
  
  ```
  (print_string "hello world\n") : unit
  ```
- Functions that create a sound, take a picture, or use a device
- Functions that raise an exception
- Functions that mutate a data 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.
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.” Google them for yourselves or 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 \( t1 \times t2 \)
  - use let \((x,y) = \ldots\) 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
What error do you get when you try to compile this file? (Type it in.) Why?

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

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

let get_name x = x.name

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

let _ = print_endline (get_name myphone)