CS 320: Concepts of Programming Languages

Lecture 4: Union and Product Types

Ankush Das

Nathan Mull

Sep 12, 2024

Administrivia

- OCaml book complements these lectures, reading the book is a must! Not all topics can be covered in the lectures
- HW1 is due today, Thursday, Sep 12, 11:59pm
- No late submissions allowed; so please submit on time
- HW2 will be released today
- Due next Thursday, Sep 19, 11:59pm
- Sooner you start, the better!

Today's Abstraction

- OCaml provides a powerful *tool* to programmers as an *abstraction*: create their own types (often called Abstract Data Types)
- This feature helps programmers define their own data structures like lists, trees, stacks, queues, etc.
- Study how to create objects of a user-defined type (*constructor*)
- Also study how to use objects of such a type (*destructor*)
- Similar to defining *structs* in C or *classes* in C++/Java but much more powerful

Topics We Will Learn Today

- Defining and Creating Union Types using Constructors
- Using Union Types using Pattern Matching
- Data-Carrying Variants
- Tuples (aka Unlabeled Product Types)
- Records (aka Labeled Product Types)
- Accessing Record Fields using Dot Notation
- Updating Records

Union Types

- Also known as *enums, variants, sum types*
- Defines objects whose values can be of different but fixed types
- e.g., defining a list whose elements can be `ints` or `bools`
- How do we define such a type? First Example: define a type called `shape` that can be either circle or square or rectangle?

```
type shape =
  | Circle
  | Square
  | Rectangle
```

Syntax for Union Types

- As usual, we will study each abstraction using syntax, type system, and semantics
- The type name is "`shape`"; it is called *variant*
- "`Circle`", "`Square`", "`Rectangle`" are called *tags* or *constructors*
- Type names are *lowercase*; constructors are *uppercase*

```
type shape =
  | Circle
  | Square
  | Rectangle
```

Syntax for Union Types

- As usual, we will study each abstraction using syntax, type system, and semantics
- The type name is "`shape`"; it is called *variant*
- "`Circle`", "`Square`", "`Rectangle`" are called *tags* or *constructors*
- Type names are *lowercase*; constructors are *uppercase*

```
type shape =
  | Circle
  | Square
  | Rectangle
```

Type name is lowercase

Syntax for Union Types

- As usual, we will study each abstraction using syntax, type system, and semantics
- The type name is "`shape`"; it is called *variant*
- "`Circle`", "`Square`", "`Rectangle`" are called *tags* or *constructors*
- Type names are *lowercase*; constructors are *uppercase*

```
type shape =
  | Circle
  | Square
  | Rectangle
```

Type name is lowercase

Constructors are lowercase

Formal Syntax for Union Types

- *Formal Syntax:*
  `type <tpname> = <Const> | <Const> | ……… | <Const>`
- `tpname` is a lowercase string; represents type name
- `Const` is an uppercase case string; represents constructor or tag
- *Type System:* none (for now); later, we will see how this type definition is added to a global signature
- *Semantics:* none; there's nothing to execute in a type definition as there's nothing to evaluate

Using Union Types

- Let's define a function `is_circle : shape -> bool` that returns `true` if the shape is `Circle` and false otherwise

- We will use pattern matching for this! That's why pattern match is such an important abstraction; *they work on any type in OCaml*

-
```
let is_circle s =
match s with
| Circle -> true
| Square -> false
| Rectangle -> false
```

Formal Syntax of Pattern Matching

- *Formal Syntax:*

```
match <expr> with
| <pattern> -> <expr>
| <pattern> -> <expr>
| ......
```

- `expr` is an expression

- `pattern` is not a value or expression but has a separate syntax (does behave like expressions)

Formal Syntax of Patterns


```
pattern ::= value-name
        |   _
        |   constant
        |   pattern as value-name
        |   ( pattern )
        |   ( pattern : typexpr )
        |   pattern | pattern
        |   constr pattern
        |   `tag-name pattern
        |   #typeconstr
        |   pattern { , pattern }+
        |   { field [: typexpr] [= pattern]{ ; field [: typexpr] [= pattern] } [; _ ] [ ; ] }
        |   [ pattern { ; pattern } [ ; ] ]
        |   pattern :: pattern
        |   [| pattern { ; pattern } [ ; ] |]
        |   char-literal .. char-literal
        |   lazy pattern
        |   exception pattern
        |   module-path .( pattern )
        |   module-path .[ pattern ]
        |   module-path .[| pattern |]
        |   module-path .{ pattern }
```

- Patterns are *typed templates* for how data of a given type can look
- They include constants, variables, wildcards, and many more
- We will not cover all patterns right now; you will gradually see them in the course

Typing a Pattern Match (Intuition)


- ```
  match e with
  | p1 -> e1
  | p2 -> e2
  | ……
  | pn -> en
  ```

- This is a generalization of if-expression
- `e` and p_i's must have the same type, say τ
- Each branch e_i must have the same type, say τ'
- Then, the whole expression has type τ'
  ```

Formal Typing Rule for Pattern Match

$$\frac{\Gamma \vdash e : \tau \qquad \forall i.\, \Gamma \vdash p_i : \tau \qquad \forall i.\, \Gamma \vdash e_i : \tau'}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n : \tau'}$$

- First premise: `e` has type τ
- Second premise: each pattern `pi` has type τ
- Third premise: Each expression `ei` has type τ'
- Conclusion: the match-expression has type τ'
- Note: we have not described how patterns are typed, will come soon

Semantics of Pattern Match (Intuition)


- ```
  match e with
  | p1 -> e1
  | p2 -> e2
  | ......
  | pn -> en
  ```
- First: evaluate expression `e`, say it has value `v`
- Match `v` with one of the patterns *in-order*, say p_i is the first pattern that matches
- Evaluate expression e_i, say its value is v_i
- Then, the whole match expression has value v_i
  ```

Formal Semantics Rule for Pattern Match

$$\frac{e \Downarrow v \qquad v = p_i \qquad \forall j < i.\, v \neq p_j \qquad e_i \Downarrow v_i}{\text{match } e \text{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n \Downarrow v_i}$$

- First premise: `e` evaluates to value `v`
- Second/Third premises: `v` matches *first* pattern p_i, none before
- Fourth premise: e_i evaluates to v_i
- Conclusion: the match-expression evaluates to v_i

Formal Semantics Rule for Pattern Match

v matches **p**$_i$

$$\frac{e \Downarrow v \qquad v = p_i \qquad \forall j < i.\, v \neq p_j \qquad e_i \Downarrow v_i}{\text{match } e \text{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n \Downarrow v_i}$$

- First premise: `e` evaluates to value `v`
- Second/Third premises: `v` matches *first* pattern p_i, none before
- Fourth premise: e_i evaluates to v_i
- Conclusion: the match-expression evaluates to v_i

Formal Semantics Rule for Pattern Match

$\mathtt{v}$ matches $\mathtt{p_i}$

$\mathtt{v}$ does not match any previous $\mathtt{p_j}$

$$\frac{e \Downarrow v \qquad v = p_i \qquad \forall j < i.\, v \neq p_j \qquad e_i \Downarrow v_i}{\text{match } e \text{ with } p_1 \to e_1 \mid \ldots \mid p_n \to e_n \Downarrow v_i}$$

- First premise: $\mathtt{e}$ evaluates to value $\mathtt{v}$
- Second/Third premises: $\mathtt{v}$ matches *first* pattern $\mathtt{p_i}$, none before
- Fourth premise: $\mathtt{e_i}$ evaluates to $\mathtt{v_i}$
- Conclusion: the match-expression evaluates to $\mathtt{v_i}$

Data-Carrying Variants

- Another cool feature of OCaml is that variant types can carry data inside the constructors.

- Recall the shape type: what if we wanted to store dimensions of the shape inside?

```
type shape =
  | Circle of int
  | Square of int
  | Rectangle of int * int
```

Data-Carrying Variants

- Another cool feature of OCaml is that variant types can carry data inside the constructors.
- Recall the shape type: what if we wanted to store dimensions of the shape inside?

```
type shape =
   | Circle of int
   | Square of int
   | Rectangle of int * int
```

Radius

Data-Carrying Variants

- Another cool feature of OCaml is that variant types can carry data inside the constructors.

- Recall the shape type: what if we wanted to store dimensions of the shape inside?

```
type shape =
  | Circle of int
  | Square of int
  | Rectangle of int * int
```

Radius

Edge

Data-Carrying Variants

- Another cool feature of OCaml is that variant types can carry data inside the constructors.
- Recall the shape type: what if we wanted to store dimensions of the shape inside?

```
type shape =
  | Circle of int
  | Square of int
  | Rectangle of int * int
```

Radius

Edge

Length & Width

Constructing Shapes

- To construct a circle of radius 5, we use the following syntax:

```
shape
let x = Circle 5
```

- To pattern match, we introduce variables for the constructor arguments. Suppose we want to compute the area of a shape.

```
shape -> float
let area s =
  match s with
  | Circle(r) -> Float.pi *. float_of_int r *. float_of_int r
  | Square(s) -> float_of_int s *. float_of_int s
  | Rectangle(l, b) -> float_of_int l *. float_of_int b
```

Why are Variants also called Unions?

- Suppose you want to define a type that can be integer or string.
- We can use variants to define such a type

```
type int_or_string =
   | Integer of int
   | String of string
```

- We can use this to define a list which stores elements that can be either integers or strings.

```
type mylist = int_or_string list
```

Topics We Will Learn Today

- Defining and Creating Union Types using Constructors
- Using Union Types using Pattern Matching
- Data-Carrying Variants
- Tuples (aka Unlabeled Product Types)
- Records (aka Labeled Product Types)
- Accessing Record Fields using Dot Notation
- Updating Records

What are Tuples?

- Tuples are *ordered fixed-length* collections of data
- The different components of a tuple are not assigned a label (unlike records, that come next)
- For example:

```
int * int
let point = (1, 2)

string * string * float * int
let record = ("Ankush", "Professor", 31.5, 320)
```

- Used to return multiple values from a function; pack data together

Describing Tuples Formally

- We study tuples the same way we study all programming abstractions
- *Syntax:*
  `(e1, e2, …, en)` — $(e_1, e_2, \ldots, e_n)$
- e.g.

```
(3 + 4, 3. +. 4.)

("A", f 2, fib (2 + 3))
```

Typing Rule for Tuples (Intuition)

- Suppose the tuple is written as $(e_1, e_2, \ldots, e_n)$
- Each component in a tuple has a type; suppose e_i has type τ_i
- Then the tuple has type $\tau_1 * \tau_2 * \ldots * \tau_n$
- Recall the types of previous examples

```
int * int
let point = (1, 2)

string * string * float * int
let record = ("Ankush", "Professor", 31.5, 320)
```

Formal Typing Rule for Tuples

$$\frac{\forall i \in [1..n].\, \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, e_2, \ldots, e_n) : \tau_1 * \tau_2 \ldots * \tau_n}$$

- **Premise: each e_i has type τ_i**
- **Conclusion: the tuple has type τ_1 * τ_2 * … * τ_n**
- **Note: Context Γ is the same for all components**

Semantics Rule for Tuples (Intuition)

- Suppose the tuple is written as `(e1, e2, …, en)` ($e_1, e_2, \ldots, e_n$)
- Evaluate each component of the typle; suppose e_i evaluates to v_i
- Then the tuple has value $(v_1, v_2, \ldots, v_n)$
- Recall the values of previous examples

```
int * int
let point = (1, 2)

string * string * float * int
let record = ("Ankush", "Professor", 31.5, 320)
```

Formal Semantics Rule for Tuples

$$\frac{\forall i \in [1..n].\, e_i \Downarrow v_i}{(e_1, e_2, \ldots, e_n) \Downarrow (v_1, v_2 \ldots, v_n)}$$

- **Premise: each e_i evaluates to value v_i**
- **Conclusion: the tuple has value $(v_1, v_2, \ldots, v_n)$**

Using Tuples

- We have already seen how to construct tuples
- Now, let's see how tuples can be used (there are multiple ways)
- Suppose we want to compute the Euclidean distance between two points: (x_1, y_1) and (x_2, y_2)
- Remember the distance is: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Option 1: Using Let


```
int * int -> int * int -> float
let dist p1 p2 =
  let (x1, y1) = p1 in
  let (x2, y2) = p2 in
  sqrt (float_of_int ((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)))
```

- *Formal Syntax:*
  `let (<varname>, <varname>, …, <varname>) = <expr> in <expr>`

Option 2: Using Simultaneous Match


```
int * int -> int * int -> float
let dist p1 p2 =
  match p1, p2 with
  | (x1, y1), (x2, y2) -> sqrt (float_of_int ((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)))
```

- *Formal Syntax:*

```
match <expr>, <expr>, … with
| (<varname>, …, <varname>),
  (<varname>, …, <varname>)
  ……… -> <expr>
```

Typing Rule for Let with Tuples

$$\frac{\Gamma \vdash e_1 : \tau_1 * \tau_2 \ldots * \tau_n \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash e_2 : \tau'}{\Gamma \vdash \text{let } (x_1, x_2, \ldots, x_n) = e_1 \text{ in } e_2 : \tau'}$$

- **First premise: e_1 must be a tuple, needs to have type $\tau_1 * \tau_2 * \ldots * \tau_n$**
- **Second premise: Add x_i with type τ_i to context and type e_2**
- **Suppose e_2 has type τ'**
- **Conclusion: Then the let-expression has type τ'**

Semantics Rule for Let with Tuples

$$\frac{e_1 \Downarrow (v_1, v_2, \ldots, v_n) \qquad [v_1/x_1, v_2/x_2, \ldots, v_n/x_n]e_2 \Downarrow v}{\text{let } (x_1, x_2, \ldots, x_n) = e_1 \text{ in } e_2 \Downarrow v}$$

- First premise: Evaluate e_1; since it's a tuple, must evaluate to a tuple with n components, say $(v_1, v_2, \ldots, v_n)$
- Second premise: Substitute x_i with corresponding value v_i and evaluate e_2
- Suppose e_2 evaluates to v
- Conclusion: Then the let-expression has value v

Topics We Will Learn Today

- Defining and Creating Union Types using Constructors
- Using Union Types using Pattern Matching
- Data-Carrying Variants
- Tuples (aka Unlabeled Product Types)
- Records (aka Labeled Product Types)
- Accessing Record Fields using Dot Notation
- Updating Records

What are Records?

- Records are *unordered fixed-length* collections of *named* data
- Each component of a record is assigned a *label*
- For example:

```
type coordinate = {x : int; y : int}

coordinate
let origin = {x = 0; y = 0}
```

- Useful for organizing large collections of data (like database records)

Formal Syntax of Records

- *Type Syntax:*
  `type <tpname> =` $\{$`f`$_1$ `:` τ_1`, f`$_2$ `:` τ_2`, …, f`$_n$ `:` τ_n`)`

- *Expression Syntax:*
  $\{$`f`$_1$ `= e`$_1$`, f`$_2$ `= e`$_2$`, …, f`$_n$ `= e`$_n$$\}$

Using Records

- We have already seen how to construct records
- Let's compute the distance using records instead of tuples
- Some people may find this cool: *records support dot-notation*

```
coordinate -> coordinate -> float
let dist p1 p2 =
  sqrt (float_of_int ((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y)))
```

- For a record `r`, field `f` can be using `r.f`
- We can also use pattern matching but we don't need to

Updating Records

- To update just one/some field(s) of record, there's a special *with-syntax*
- Let's see some examples:

```
coordinate -> coordinate
let proj_x p = { p with y = 0}

coordinate -> coordinate
let inc_x p = { p with x = p.x + 1 }
```

- Read this as: *update* `p` *"with … and keep the rest the same"*
- Note that records are immutable by default, so calling these functions creates a new record

Homework

- Read about records
- Write formal typing and semantics rules for records
- Practice as many typing derivations and semantics derivations as you can! Please!
- Write typing rules for other pattern-match for tuples
- Read OCaml Book 3.2, 3.4, 3.5