Variants and patterns

EFC :: W3L1

Dan R. Ghica

2017

“Variant” types

```
type weekday = Mon | Tue | Wed | Thu
             | Fri | Sat | Sun
```

Suppose that I want to produce a **string** describing how I feel on a particular **weekday.**

Pattern matching

```
let how_i_feel day =
   match day with
    | Mon → "sleepy"
    | Tue → "grumpy"
    | Wed → "sneezy"
    | Thu → "dopey"
    | Fri → "happy"
    | Sat → "bashful"
    | Sun → "doc";;
val how_i_feel : weekday → string = <fun>
```

A **key** feature

Syntax

```
match expression with
| case1 -> expression1
| case2 -> expression2
| …
| casen -> valuen
```

Evaluation

match *case*$_k$ **with**

| …

| *case*$_k$ -> *expression*$_k$

| …

becomes just

expression$_k$

```
# how_i_feel Tue;;
- : string = "grumpy"
# how_i_feel "Tue";;
Characters 11-16:
  how_i_feel "Tue";;
             ^^^^^
Error: This expression has type string but an
expression was expected of type
         weekday
#
```

```
how_i_feel Tue
⟶
let day = Tue in
match day with
  | Mon → "sleepy"
  | Tue → "grumpy"
  | Wed → "sneezy"
  | Thu → "dopey"
  | Fri → "happy"
  | Sat → "bashful"
  | Sun → "doc"
⟶
match Tue with
  | Mon → "sleepy"
  | Tue → "grumpy"
  | Wed → "sneezy"
  | Thu → "dopey"
  | Fri → "happy"
  | Sat → "bashful"
  | Sun → "doc"
⟶
"grumpy"
```

Using ‘`if`’ statements

```
let how_i_feel day =
  if day = Mon then "sleepy"
  else if day = Tue then "grumpy"
  else if day = Wed then "sneezy"
  else if day = Thu then "dopey"
  else if day = Fri then "happy"
  else if day = Sat then "bashful"
  else if day = Sun then "doc"
  else "error";;
```

A comparison

verbose

```
let how_i_feel day =
  if day = Mon then "sleepy"
  else if day = Tue then "grumpy"
  else if day = Wed then "sneezy"
  else if day = Thu then "dopey"
  else if day = Fri then "happy"
  else if day = Sat then "bashful"
  else if day = Sun then "doc"
  else "error";;
```

redundant **else**

vs

```
let how_i_feel day =
  match day with
    | Mon → "sleepy"
    | Tue → "grumpy"
    | Wed → "sneezy"
    | Thu → "dopey"
    | Fri → "happy"
    | Sat → "bashful"
    | Sun → "doc";;
```

A more significant advantage

```
# let how_i_feel day =
  match day with
    | Mon → "sleepy"
    | Tue → "grumpy"
    | Wed → "sneezy"
    | Thu → "dopey"
    | Fri → "happy"
    | Sat → "bashful";;
              Characters 24-152:
  ..match day with
    | Mon → "sleepy"
    | Tue → "grumpy"
    | Wed → "sneezy"
    | Thu → "dopey"
    | Fri → "happy"
    | Sat → "bashful"..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Sun
val how_i_feel : weekday → string = <fun>
```

A more significant advantage

```
# let how_i_feel day =
  match day with
    | Mon → "sleepy"
    | Tue → "grumpy"
    | Wed → "sneezy"
    | Thu → "dopey"
    | Fri → "happy"
    | Sat → "bashful"
    | Sun → "doc"
    | Mon → "sadface";;
                    Characters 172-175:
    | Mon → "sadface";;
      ^^^
Warning 11: this match case is unused.
val how_i_feel : weekday → string = <fun>
```

PM and types

- Pattern matching allows the compiler to **type-check** your case statement for **exhaustiveness** and **reachability**!

- Avoid a significant source of **runtime** errors

Comparison with Java

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
public class EnumTest {
    public String how_i_feel(Day day) {
        switch (day) {
            case MONDAY:    return "sleepy";
            case TUESDAY:   return "grumpy";
            case WEDNESDAY: return "sneezy";
            case THURSDAY:  return "dopey";
            case FRIDAY:    return "happy";
            case SATURDAY:  return "bashful";
            case SUNDAY:    return "doc";
        }
        return "unreachable";
    }
```

vestigial C

boilerplate

no type-checking

Function by cases

Abbreviation:

```
let how_i_feel day = match day with
                   | Monday → …

let how_i_feel = function
               | Monday → …
```

more succinct

```
# let how_i_feel = function
    | Mon → "sleepy"
    | Tue → "grumpy"
    | Wed → "sneezy"
    | Thu → "dopey"
    | Fri → "happy"
    | Sat → "bashful"
    | Sun → “doc";;
val how_i_feel : weekday → string = <fun>
```

Some cool (?) “functional” stuff

Alternative notations for application

- If you find
  `sqrt (dbl (sqr (1.0)))`
  too ugly (too brackety and inside-outy) you can try

- `sqrt @@ dbl @@ sqr @@ 1.0 ;;`
  which associates to the **right**

- `1.0 |> sqr |> dbl |> sqrt ;;`
  which looks like a **pipeline**
  and associates to the **left**

Tuples

(a, b)

Recall

```
let h (a, b) = sqrt(a *. a +. b *. b);;

val h : float * float → float = <fun>
```

"product" = the argument is a *pair*

```
# h (2.0, 3.0);;
- : float = 2.8284271247461903
```

Tuples are themselves data

```
let h (a, b) = sqrt(a *. a +. b *. b);;
val h : float * float → float = <fun>

# h (2.0, 3.0);;
- : float = 2.8284271247461902 9

# let x = (2.0, 3.0);;
val x : float * float = (2., 3•)

# h x;;
- : float = 2.8284271247461902 9
```

Remember rewriting

```
let x = (2.0, 3.0)
```

```
h x → h (2.0, 3.0) → …
```

Projections

```
# let fst (x, y) = x;;
val fst : 'a * 'b → 'a = <fun>

# let snd (x, y) = y;;
val snd : 'a * 'b → 'b = <fun>

# let hyp ab =
  sqrt ((fst ab) *. (fst ab)
     +. (snd ab) *. (snd ab));;
val hyp : float * float → float = <fun>
```

Pattern matching 🔥🍰

```
# let x = (2.0, 3.0);;
val x : float * float = (2.0, 3.0)
```

```
# let (a, b) = (2.0, 3.0);;
val a : float = 2.0
val b : float = 3.0

# let (a, b) = x;;
val a : float = 2.0
val b : float = 3.0
```

Pattern matching

```
# let (a, b) = (2.0, 3.0) in a +. b;;
- : float = 5.
```

```
# match (2.0, 3.0) with
  | (a, b) → a +. b ;;
- : float = 5.
```

Pattern matching and variables

```
# match (2.0, 3.0) with
        | (a   , b   ) → a +. b ;;
- : float = 5.
```

- **match** the constant elements
- **define** the variables on the left of the arrow

Pattern matching and variables

```
# match (2.0, 3.0) with
      | (2.0, b  ) → b +. b ;;
Characters 0-44:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(0., _)
- : float = 6.
```

- **match** the constant elements
- **define** the variables on the left of the arrow

Pattern matching and variables

```
# match (1.0, 3.0) with
        | (2.0, b  ) → b +. b ;;
Characters 0-44:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(0., _)
Exception: Match_failure ("//toplevel//", 1, 0).
```

- **match** the constant elements
- **define** the variables on the left of the arrow

Pattern matching and variables

```
# match (1.0, 3.0) with
      | (2.0, b  ) → b +. b
      | (a  , 3.0) → a +. a;;
Characters 0-44:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(0., 0•)
- : float = 2.
```

- **match** the constant elements
- **define** the variables on the left of the arrow

Pattern matching and variables

```
# match (1.0, 3.0) with
      | (2.0, b  ) → b +. b
      | (a  , 3.0) → a +. a
      | (a  , b  ) → a +. b;;
- : float = 2.
```

- match the constant elements
- define the variables on the left of the arrow
- **go top-to-bottom**

No multiple occurrences of variables in the pattern

```
utop # match (1.0, 3.0) with
         | (a  , a) -> a +. a;;
Error: Variable a is bound several times in this
matching
```

No expressions in the pattern

```
utop # match (1.0     , 3.0) with
         | (a ± 1.0, b) -> a +. b;;
Error: Syntax error: ')' expected, highlighted '(' might be unmatched
```

Special feature: ignored variable

```
# match (1.0, 3.0) with
        | (2.0, b  ) → b +. b
        | (a  , 3.0) → a +. a
        | _          → 0. ;;  (* Default 💩 *)
- : float = 2.
```

- match the constant elements
- define the variables on the left of the arrow
- **go top-to-bottom**

Tuples in Java

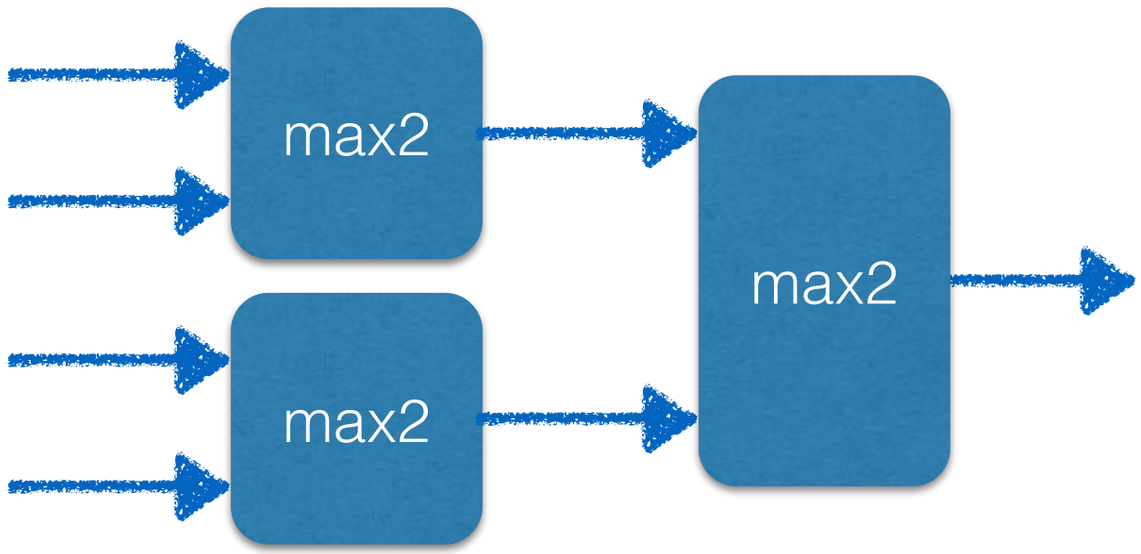
```
public class Tuple<X, Y> {
   public final X x;
   public final Y y;
   public Tuple(X x, Y y) {
      this.x = x;
      this.y = y; }
}
public Integer
hyp (Tuple<Float, Float> xy)
{ return sqrt(xy.x * xy.x
              + xy.y * xy.y); }
```

Programming with tuples

```
let max2 (a, b) = if a > b then a else b;;


let max3 (a, b, c) =
    if a ⩾ b && b ⩾ c then a
    else if a ⩾ c && c ⩾ b then a
    else if b ⩾ a && a ⩾ c then b
    else if b ⩾ c && c ⩾ a then b
    else if c ⩾ a && a ⩾ b then c
    else if c ⩾ b && b ⩾ a then c


let max4 (a, b, c, d) = ???
```

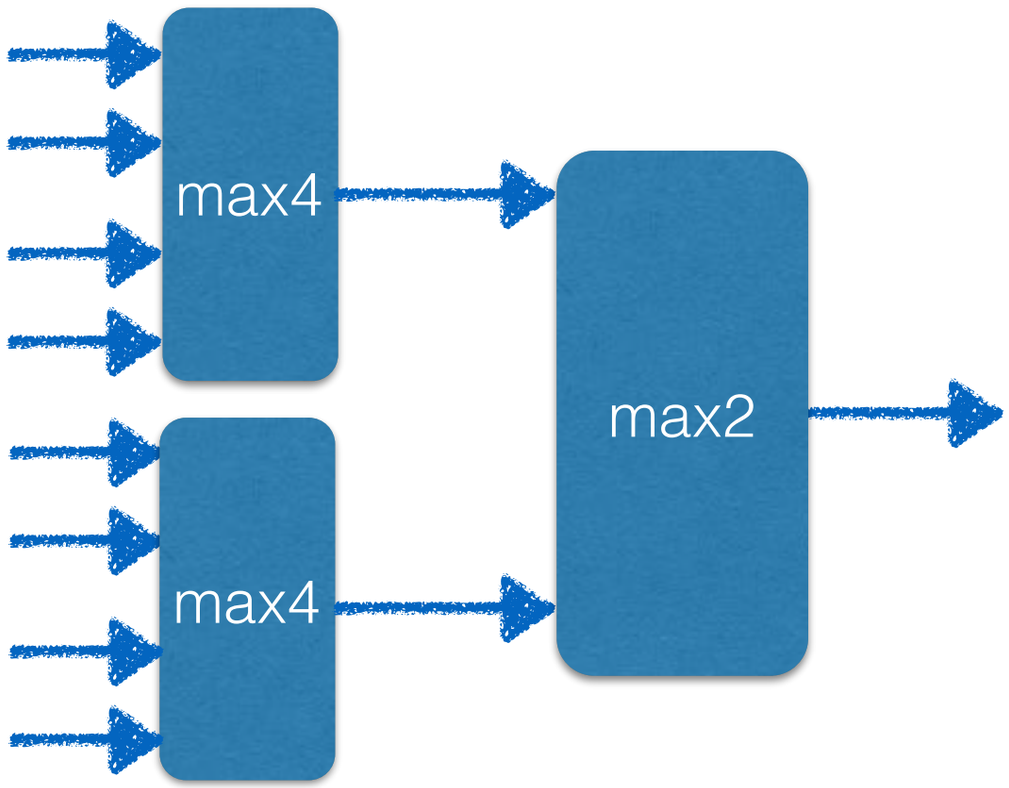

max2
max2
max2

```
let max2 (a, b) = if a > b then a else b;;

let max3 (a, b, c) = … (* ? *)
        max2 (a, max2 (b, c))

let max4 (a, b, c, d) = … (* ? *)
        let m1 = max2 (a, b) in
        let m2 = max2 (c, d) in
        max2(m1, m2)

let max8 (a1, a2, a3, a4, a5, a6, a7, a8) =
        let m1 = max4 (a1, a2, a3, a4) in
        let m2 = max4 (a5, a6, a7, a8) in
        max2 (m1, m2)
```


max4
max4
max2

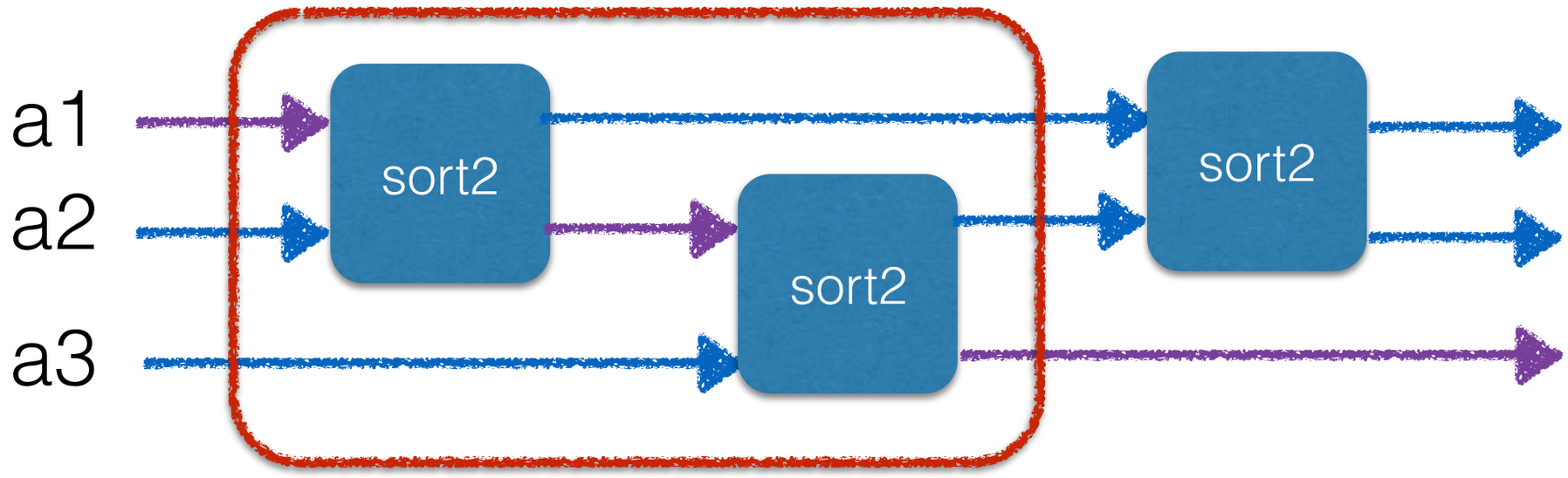
You could (?) also write a big **if** statement… but don’t!

Use **functions**!

```
let sort2 (a, b) = if a < b
                   then (a, b)
                   else (b, a);;
```

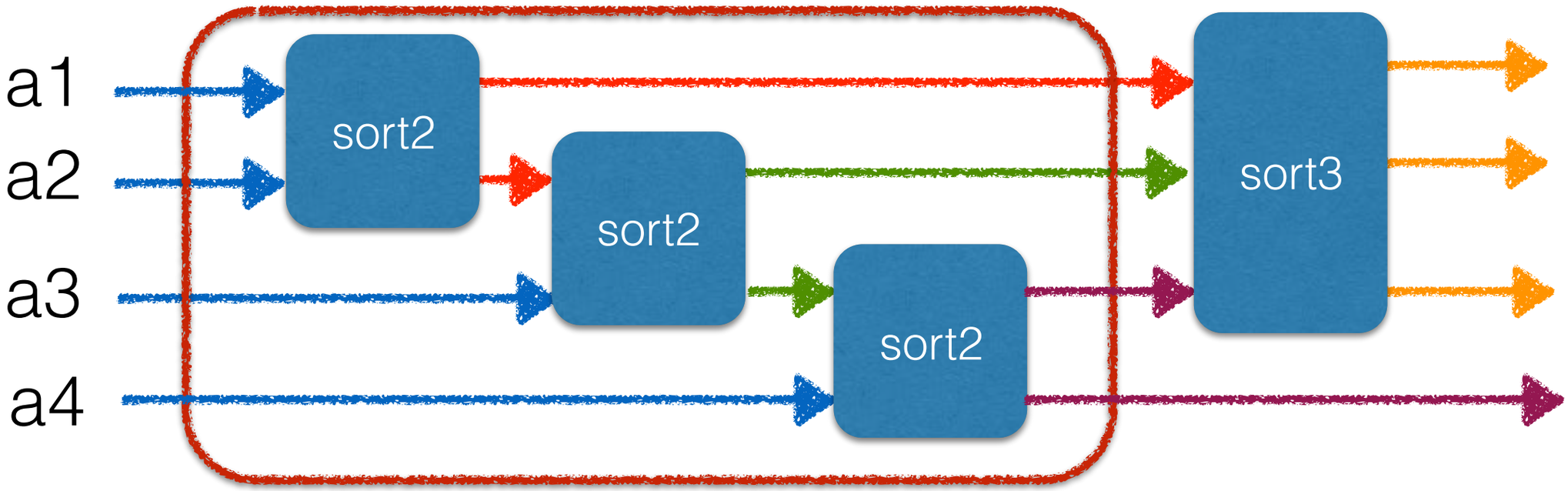
```
let sort3 (a1, a2, a3) = ???
```

How to **think** about it? Can we use this?




```
let sort3 (a1, a2, a3) =
    let (a1, a2) = sort2 (a1, a2) in
    let (a2, a3) = sort2 (a2, a3) in
    let (a1, a2) = sort2 (a1, a2) in
    (a1, a2, a3)
```

sort4




```
let sort4 (a1, a2, a3, a4) =
    let (a1, a2) = sort2 (a1, a2) in
    let (a2, a3) = sort2 (a2, a3) in
    let (a3, a4) = sort2 (a3, a4) in
    let (a1, a2, a3) = sort3 (a1, a2, a3) in
    (a1, a2, a3, a4)
```

A difficult question

- Given a tuple of size N what is the best sorting network?
  - fewest sorters
  - shortest path

“Sorting networks”

- Further reading (not examinable but v. cool)

- **Reference**: https://en.wikipedia.org/wiki/Sorting_network

- **Tutorial**: http://hoytech.github.io/sorting-networks/

Week 3 survey

http://bit.ly/focs05pp

Answers