Programovací jazyky F# a OCaml

Chapter 3.

Composing primitive types into data

Data types

» We can think of data type as a set:

int = { ... -2, -1, 0, 1, 2, ... }

More complicated with other types, but possible…

» Functions are maps between sets (math!)

For example, the function: *f : X -> Y*

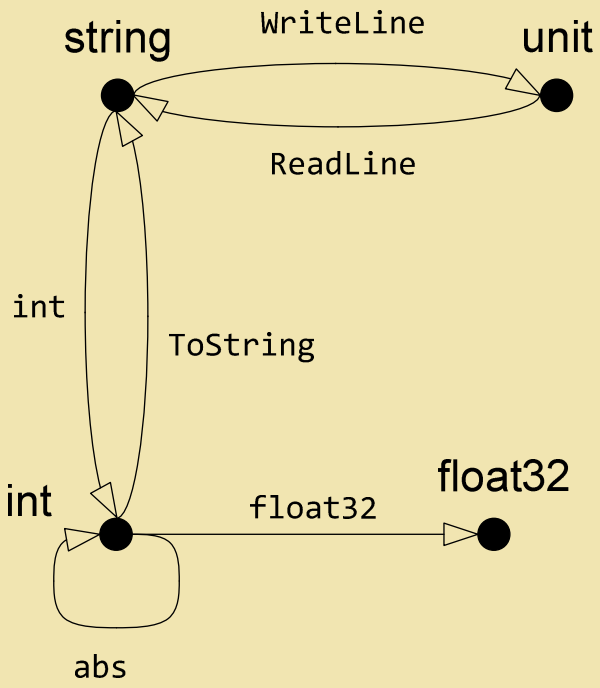
f(x) = y assigns value y ∈ Y to any x ∈ X

Functions is undefined for x ∈ X

Keeps looping forever or throws an exception

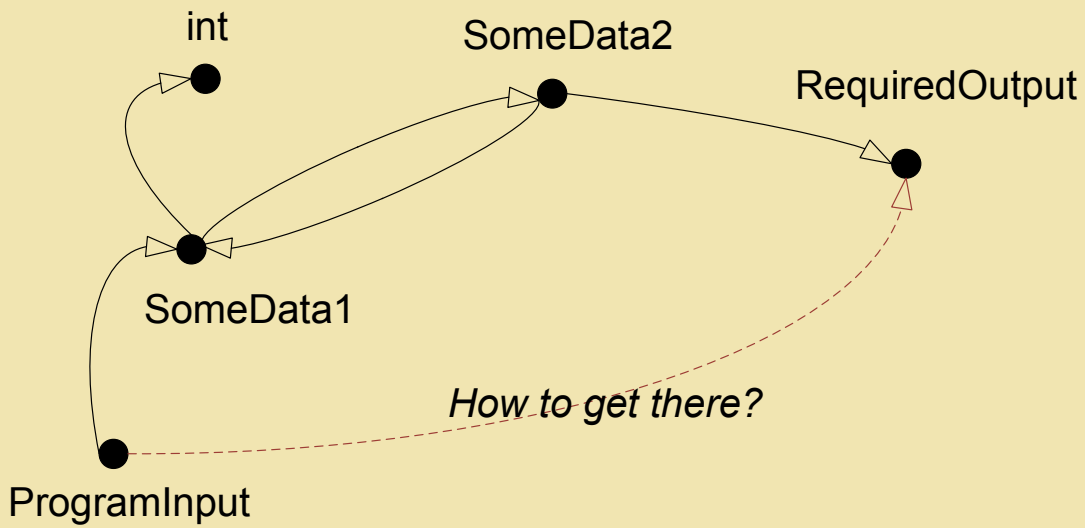
Drawing functions as diagrams

- Functions are arrows between different types
- Higher-order functions
  - Tricky – we also need type for functions
- Multi-argument functions
  - Are higher-order too



Diagrams are still useful!

» Identify the key input/output of a function



» Relations with mathematics: *category theory*

Composing data types

Operations with sets

- **Product:** $X \times Y = \{(x, y) | x \in X \text{ and } y \in Y\}$
- **Sum:** $X + Y = \{(1, x) | x \in X\} \cup \{(2, y) | y \in X\}$

- Some examples:
  - int × char = { … (-1, a), (0, a), … (-1, b), (0, b) … }
  - int ∪ char = { … -1, 0, 1, … a, b, … }
  - int + char = { … (1, -1), (1, 0), (1, 1), … (2, a), (2, b), … }

How can we distinguish between **int** and **char**?

tag

- Constructing complex types in F#
  - Two options: **Product** types and **Sum** types

Product types in F#

Tuple type (product)

» Store several values of possibly different types

The number is known at compile time

```
> let resolution = (1600, 1200);;
val resolution : int * int = (1600, 1200)

> fst resolution;;
val it : int = 1600

> match resolution with
    | (width, height) -> width * height;;
val it : int = 1920000
```

Two values of type int (*one value of int × int*)

fst and **snd** return components of two-component tuple

Decomposing tuple using pattern matching

pattern

» We can use pattern matching without **match**!

Pattern matching for tuples

- Pattern matching in **let** binding

```
> let (width, height) = resolution;;
val width : int = 1600
val height : int = 1200

> let num, str = 12, "hello";;
val str : string = "hello"
val num : int = 12
```

pattern

We can omit parenthesis!
Binding multiple values

- The **match** construct is still important!

```
> match resolution with
  | (w, h) when float h / float w < 0.75 -> "widescreen"
  | (w, h) when float h / float w = 0.75 -> "standard"
  | _ -> "unknown";;
val it : string = "standard"
```

Tuples as parameters

- Using tuples as return values or parameters
  - Computation is written as an expression
  - We need expressions that return more values!

```
> let divMod (a, b) =
    let d = a / b
    let r = a % b
    (d, r);;
val divMod : int * int -> int * int
```

pattern

Note the difference:
int -> int -> int * int
int * int -> int * int

- Tuples as arguments - looks like standard call!

```
> let d, rem = divMod (17, 3);;
val rem : int = 2
val d : int = 5
```

Tuples

- Combine multiple values (set product)
  - Expressions that calculate multiple values
  - Specifying parameters of functions
  - Grouping logically related parameters
  - **Example**: X and Y coordinates, day + month + year
- Very simple data type
  - No information about elements (only types)
  - Avoid using too complex tuples

Record type (also product)

» Type with multiple named fields

Type needs to be declared in advance

```
open System
type Lecture =
  { Name : string
    Room : string
    Starts : DateTime }

let fsharp =
  { Name = "F# and OCaml"
    Room = "S11"
    Starts = DateTime.Parse
       ("5.10.2009 14:00") }

> fsharp.Name;;
val it : string = F# and OCaml

> match fsharp with
  | { Name = nm; Room = rm } -> printfn "%s in %s" nm rm;;
F# and OCaml in S11
```

Type declaration

Specify values (type is inferred!)

Accessing field by name

Decomposing using pattern

Calculating with records

- Records (tuples, …) are all immutable
  - How to change schedule of a lecture?
  - Calculate new value of the schedule

```
let changeRoom room lecture =
  { Name = lecture.Name; Starts = lecture.Starts
    Room = room }

let newfs changeRoom "S8" fsharp
```

Copy fields that don't change

Specify new value

- Cloning record with some change is common

```
let changeRoom room lecture =
  { lecture with Room = room }
```

Records

- Used for more complex data structures
  - Fields describe the meaning of the code
  - Easy to clone using the **with** keyword
- Tuples store *values*, Records store *data*
  - *Data* – the primary thing program works with
  - *Values* – result of an expression (intuitive difference!)
- In F#, compiled as .NET classes
  - Can be easily accessed from C#

Sum types in F#

Discriminated union (Sum)

» Data type that represents alternatives

```
type Variant =
  | Int of int
  | String of string
```

int + string =

{ ..., (1, -1), (1, 0), (1, 1), ...
..., (2, ""), (2, "a"), (2, "aa"), ... }

» More examples:

```
type Season =
  | Spring
  | Summer
  | Autumn
  | Winter
```

Simplified example – union cases do not carry any values

```
type Shape =
  | Circle of int
  | Rect of int * int
```

Using tuple as the carried value

Working with unions

- Distinguishing between cases using patterns

```
let shapeArea shape =
  match shape with
  | Circle(radius) -> Math.PI * (pown radius 2)
  | Rectangle(width, height) -> width * height
```

Discriminator

Nested pattern: extracts values

- Compile-time checking of patterns

```
match var with
| String(msg) -> printfn "%s" msg
```

Warning: Incomplete pattern match

- Pattern matching using **let**:

```
let (Int(num)) = var
```

Syntactically correct, but rarely useful

Discriminated unions

- Used to represent value with different cases
  - Expression evaluates and returns **A** or **B**
  - The compiler verifies that we handle all cases
  - Single-case unions are sometimes used
- Similar to class hierarchies in OOP
  - We can more easily add new functions
  - Adding new cases requires modifying all functions

Homework #1

» *We used "sum" of sets to model discriminated unions and "product" to model tuples:*

$$X + Y = \{(1, x) | x \in X\} \cup \{(2, y) | y \in X\}$$
$$X \times Y = \{(x, y) | x \in X \text{ and } y \in Y\}$$

» *How can we use this operations to construct mathematical model of the following types:*

```
type Season =
  | Spring
  | Summer
  | Autumn
  | Winter
```

```
type Shape =
  | Circle of int
  | Rectangle of int * int
```

F# option type

» Represent a value or an empty value

Mathematically: add missing value **int** + {⊥}

```
let opt = Some(42)
match opt with
| Some(n) ->
    printfn "%d" n
| None ->
    printfn "nothing";;

// Prints: 42
```

Correct handling of empty values

```
> let square opt =
    match opt with
    | Some(n) -> Some(n * n)
    | None -> None;;
val square : int option
           -> int option

> square (Some 4);;
val it : int option = Some 16
> square None;;
val it : int option = None
```

Takes and returns "int option"

Pattern matching

Representing complex data

» Combining tuples and unions

```
type Color =
  | Red
  | White
  | Blue

type VehicleType =
  | Bicycle  // no additional information
  | Car of int * int  // #doors * horses
  | Motorcycle of int // motor ccm

type Vehicle =
  Color * string * VehicleType
```

We could use System.Drawing.Color

Type of vehicle with specific properties

Type alias for tuple (we could use records too)

Pattern matching

```
match vehicle with
| Red, name, _ when name.StartsWith("S") ->
   printfn "some red S..... vehicle"
| _, "Mazda", Car(5, h) when h > 100 ->
   printfn "5-door Mazda with >100 horses"
| White, _, Bicycle
| White, _, Motorcycle(_) ->
   printfn "white bicycle or motorcycle"
| _, _, (Car(5, _) & Car(_, 200)) ->
   printfn "car with 5 doors and 200 horses"
| _, _, (Car(_, _) as veh) ->
   printfn "car: %A" veh
```

when clause

nested pattern

or pattern

and pattern

alias pattern

» **Other uses:** simplification or symbolic differentiation of an expression, compilation

Homework #2

» *Write a function that compares two vehicles and prints detailed information about the more expensive one.*

1. *Motorcycle is always more expensive than bicycle*
2. *White bicycles are more expensive than other bicycles*
3. *Car is always more expensive than motorcycle (with the exception of **Trabant** which is cheaper than motorcycles with engine more than **100ccm**)*
4. *The more ccm engine a motorcycle has the better*
5. *Ferrari and Porsche are more expensive than any other cars (when comparing Ferraris with Porches, the rule 6 applies)*
6. *The more horses car has the better (if horsepower equals, the more doors it has the better)*

***Bonus point**: if you'll handle all cases when the first vehicle is more expensive than the second in one match clause*