Lec 6: Data Abstraction

CS220: Programming Principles

Sang Kil Cha

Compound Data

Motivation

Can we combine primitive data types that we learned so far to represent more complex data types?

Motivation

Can we combine primitive data types that we learned so far to represent more complex data types?

What's the ***glue***?

Motivating Example: Rational Numbers

A rational number is any number that can be expressed as the quotient of two integers: p/q, where p is a numerator and q is a denominator.

Motivating Example: Rational Numbers

A rational number is any number that can be expressed as the quotient of two integers: p/q, where p is a numerator and q is a denominator.

Assume that we have the following three functions:

Motivating Example: Rational Numbers

A rational number is any number that can be expressed as the quotient of two integers: p/q, where p is a numerator and q is a denominator.

Assume that we have the following three functions:

- **Constructor:** a function that takes in two integers and returns a rational number (`makeRat`).

Motivating Example: Rational Numbers

A rational number is any number that can be expressed as the quotient of two integers: p/q, where p is a numerator and q is a denominator.

Assume that we have the following three functions:

- **Constructor:** a function that takes in two integers and returns a rational number (`makeRat`).
- **Numerator Selector:** a function that takes in a rational number and returns the numerator of the rational number (`numer`).

Motivating Example: Rational Numbers

A rational number is any number that can be expressed as the quotient of two integers: p/q, where p is a numerator and q is a denominator.

Assume that we have the following three functions:

- **Constructor:** a function that takes in two integers and returns a rational number (`makeRat`).
- **Numerator Selector:** a function that takes in a rational number and returns the numerator of the rational number (`numer`).
- **Denominator Selector:** a function that takes in a rational number and returns the denominator of the rational number (`denom`).

Writing Basic Operators for Rational Numbers

Addition ($n_1/d_1 + n_2/d_2 = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$).

```
let addRat x y =
  makeRat (((numer x) * (denom y)) + ((numer y) * (denom x)))
          (denom x * denom y)
```

Subtraction ($n_1/d_1 - n_2/d_2 = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$).

```
let subRat x y =
  makeRat (((numer x) * (denom y)) - ((numer y) * (denom x)))
          (denom x * denom y)
```

Writing Basic Operators for Rational Numbers

Multiplication ($n_1/d_1 \times n_2/d_2 = \frac{n_1 n_2}{d_1 d_2}$).

```
let mulRat x y =
  makeRat (numer x * numer y) (denom x * denom y)
```

Division ($\frac{n_1/d_1}{n_2/d_2} = \frac{n_1 d_2}{d_1 n_2}$).

```
let divRat x y =
  makeRat (numer x * denom y) (denom x * numer y)
```

Our First Glue: Tuples

A tuple is a grouping of unnamed but ordered values, possibly of different types.

Tuples.

```
(1, 2) // (int * int)
("a", "b", "c") // (string * string * string)
(1, "abc") // (int * string)
```

Define a Type

You can explicitly define a type using the `type` keyword.

```
type Point = int * int

let p = (1, 2) // This is compatible with Point.
let p: Point = (1, 2) // Can even specify the type.
```

Accessing Elements in Tuples

```
let x = (1, 2)
fst x // returns 1
snd x // returns 2
// thr x <- this doesn't exist
let y = (1, 2, 3)
let _, _, third = y // we can get the third value.
let fst (e, _) = e // matching a tuple as arg.
```

Representing Rational Numbers w/ Tuples

```
let makeRat n d = (n, d)
let numer x = fst x
let denom x = snd x
```

Can this definition handle negative rational numbers? What is the problem?

Normalization is Required

```
makeRat (-1) 2 // -0.5
makeRat 1 (-2) // -0.5
(-1, 2) = (1, -2) // false
```

Can you fix the `makeRat` function so that the same rational numbers can always have the same tuple?

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

In-Class Activity #04

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

The Problem

Modify the `makeRat` function so that the RationalNumber type is comparable.

Data Abstraction

Data Abstraction

A methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects.

For example, we don't need to know how rational numbers are constructed in order to write `addRat`, `subRat`, etc.

Our Second Glue: Records

Records aggregate “named values”.

Records.

```
type RationalNumber = { // Type definition.
  Numerator : int
  Denominator : int
}

let n = { Numerator = 2; Denominator = 3 }
n.Numerator // returns 2
n.Denominator // returns 3
```

Type Ambiguity

```
type Point = { X: float; Y: float; Z: float }
type XXXXX = { X: float; Y: float; Z: float }
let x = { X = 1.0; Y = 1.0; Z = 1.0 } // Point or XXXXX?
let x = { Point.X = 1.0; Y = 1.0; Z = 1.0 } // Be explicit.
```

Making New Record from an Existing Record

We ***cannot*** update fields of a record, but we can create a new one.

```
type Point = { X: float; Y: float; Z: float }
let p = { X = 1.0; Y = 1.0; Z = 1.0 } // (1.0, 1.0, 1.0)
let q = { p with Y = 2.0 } // (1.0, 2.0, 1.0)
let r = { p with X = 3.0; Z = 3.0 } // (3.0, 1.0, 3.0)
// p, q, r are all alive here.
```

Immutability

Data types in F# are ***immutable*** by default (thus, no side-effects). Meaning that you cannot change the value once a data object is constructed.

The fact that objects will never be mutated helps write "***reasonable***" code.

Our Third Glue: Discriminated Unions

Both records and tuples create a new type by “multiplying” types together. But what if we want to “sum” multiple types together to create a new one?

Our Third Glue: Discriminated Unions

Both records and tuples create a new type by “multiplying” types together. But what if we want to “sum” multiple types together to create a new one?

The (`int * Bool`) type can have $2^{32} * 2 = 2^{33}$ possible values. And we want to create a type that accepts only $2^{32} + 2$ possible values.

Discriminated Unions

IntOrBool.

```
type IntOrBool =
  | Int of int
  | Bool of bool

Int 42 // constructs IntOrBool
Bool false // constructs IntOrBool
```

Discriminated Unions (cont'd)

Days.

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

Selector for Discriminated Unions

How can we extract values from a discriminated union?

Pattern Matching!

Pattern Matching

Patterns

Each type in F# mostly has its own pattern.

- `_`: underscore for matching "any" type.
- `(_, _, _)`: tuples.
- `{ X = x }`: records.
- `LabelName _`: discriminated unions.
- `...`

We will see more patterns as we learn more data types.

`let`-Bindings for Patterns

```
let (Int x) = Int 42 // x has a value 42
let (a, b) = (1, "hello") // a = 1 and b = "hello"
type Point = { X: int; Y: int }
let { X = x } = { X = 1; Y = 2 } // x = 1
let _ = Int 42 // We effectively ignore the value.
```

Matching Values

Pattern matching.

```
match e with
| PatternA -> eA
| PatternB -> eB
| ... // omitted
```

First evaluate `e` and match the evaluated value with the following patterns. If it matches `PatternA`, then evaluate `eA`. Else, if it matches `PatternB`, then evaluate `eB`. And so on and so forth.

Why not just use `if then else`?

You can do it, but pattern matching is much more elegant!

```
// With pattern matching.
match x with
| (0, 0) -> "a"
| (1, _) -> "b"
| (_, _) -> "c"

// With if-then-else
if fst x = 0 && snd x = 0 then "a"
elif fst x = 1 then "b"
else "c"
```

Q: What's the Result?

```
let filter x =
  match x with
  | num -> "others"
  | 1 | 2 | 3 -> "1 or 2 or 3"
filter 4 // ?
```

Pattern Matching with Guards

We can add a `when` clause right next to each pattern in a pattern matching expression to specify an additional condition to match (a ***guard***).

```
let rangeTest v =
  match v with
  | v when v >= 0 && v < 42 -> true
  | _ -> false
```

Make it Simpler

The `function` keyword, which represents a function taking in only a single argument, can be used for pattern matching with out the use of `match` keyword.

Rewriting the previous example `rangeTest`.

```
let rangeTest = function
  | v when v >= 0 && v < 42 -> true
  | _ -> false
```

Function vs. Pattern Match

- A pure function maps a value in a set to a value in another set.
- A match statement is the same!

> You can always define a function with a pattern matching.

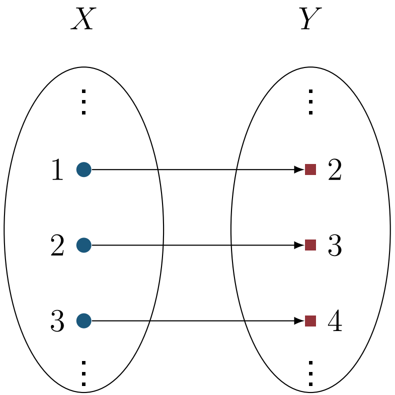
Should We Always Use Pattern Matching for Defining Functions?

No. Consider the following case.

Example: `addByOne`

```
let addByOne = function
  | 1 -> 2
  | 2 -> 3
  | ...
```

In Math ...



Quick Exercise

```
let rec factorial n =
  if n <= 1 then 1
  else n * factorial (n - 1)
```

Re-write the `factorial` function using the `function` keyword. Do you think it is better than the above one? Why or why not?

Function Arguments

A function that takes two integers as input:

```
let sumA a b = a + b
let sumB (a, b) = a + b
```

In-Class Activity #05

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

The Problem

Consider a world with only three shapes: circle, square, and triangle.

```
type Shape =
  /// A circle of a radius.
  | Circle of float
  /// A square with a side length.
  | Square of float
  /// A triangle with side lengths.
  | Triangle of float * float * float
```

Modify the `area` function, which computes the area of a given shape.
Hint: Heron's Formula is

$$Area(a, b, c) = \sqrt{p(p-a)(p-b)(p-c)}, \text{ where } p = \frac{a+b+c}{2}.$$

Conclusion

Algebraic Data Types

We can "add" or "multiply" data types to combine them into a new data type.

Kind	Our Glue	Meaning
Product types	***tuples***, ***records***	A and B and ...
Sum types	***discriminated unions***	A or B or ...

F# Naming Convention

Haven't explicitly mentioned yet, but there is a common naming convention that you want to follow in F#.

1. Use `camelCase`[1] for values (including functions).
2. Use `PascalCase` for types (including modules and classes).

[1] We sometimes call `camelCase` as lower camel case, and `PascalCase` as upper camel case.

Question?

Further Reading

- The wizard book: Chapter 2.1.
- `https://fsharpforfunandprofit.com/posts/discriminated-unions/`
- `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching`