Functional Programming

Records, Lists and Modelling

Aske W. Brekling

awb@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark

Overview

In the lecture room:

- Records: Patterns, Selectors, Functions, Type Declarations
- Modelling: CD Register - Use of record and list patterns
- Modelling: Cash Register - Problem Solving

In the G-databar:

- Exercise 6.2 - Dating Bureau
- Exercise 6.3 - Map-colouring

Records: Declarations, Selectors

Record declaration - $\{label_1 = value_1, label_2 = value_2, ...\}$

```
- val a = {name = "Peter", age = 20};
> val a = {age=20,name="Peter"} :
          {age:int, name:string}
```

The record a contains the string $Peter$ with label $name$, and the integer 20 with label age.

Record selector - $\#label_i\ record$

```
- #name a;
> val it = "Peter" : string
```

Selects the value from the record a with label $name$

Records: Patterns

Record patterns - $\{label_1 = val_1, label_2 = val_2, ...\}$

```
- val {name = x, age = y} = a;
> val x = "Peter" : string
  val y = 20 : int
```

Patterns are used to decompose a record into its components

Short form record patterns - $\{label_1, label_2, ...\}$

```
- val {name, age} = a;
> val name = "Peter" : string
  val age = 20 : int
```

Used instead of:

```
- val{name = name, age = age} = a;
> val name = "Peter" : string
  val age = 20 : int
```

Records: Functions

Equality - $\text{record}_1 = \text{record}_2$

```
- {age = 20, name = "Peter"} =
  {name = "Peter", age = 20};
> val it = true : bool
```

Equality of records with the same type is defined component-wise. Order has no importance. Comparison only allowed for same-type records.

Wild card - ...

```
- val {name = x, ...} = a;
> val x = "Peter" : string
```

Record patterns may contain some of the labels only - used when only some components are needed. Useful for handling data with many components and functions only using a fraction of them.

Records: Type Declarations

Type declaration - $\{label_1 : type_1, label_2 : type_2, ...\}$

```
- type person = {age : int, birthday : int * int,
  name : string, occupation : string, sex : string };
> type person = {age : int, birthday : int * int,
  name : string, occupation : string, sex : string}
```

Data for a person represented by the record type `person`

Functions on records

```
- fun age(p: person) = #age p;
> val age = fn : {age : int, birthday : int * int,
  name : string, occupation : string, sex : string}
  -> int
```

Modelling example: CD Register

We want to model a register describing CDs. Each CD is described by its $title$, $artist$, $record\ company$, $year$ and the $songs\ on\ the\ disc$.

We might want to construct functions only using some of the components, therefore modelling CDs as records with the aforementioned components would be good. We name this record type: cd

It makes sense to model the $title$, $artist$ and $company$ components as strings, the $year$ as an integer and the $songs$ as a string list.

The full register is modelled as a cd list

Modelling example: CD Register (decl.)

Type declaration of CD registers:

```
type cd = {title: string, artist: string,
           company: string, year: int,
           songs: string list};

type cdRegister = cd list;
```

Example of a CD register:

```
val cdreg = [{title="t1", artist="a1", company="c1",
              year=93, songs=["s1","s2","s3","s4"]},
             {title="t2", artist="a2", company="c2",
              year=91, songs=["s6","s7","s8","s9"]},
             {title="t3", artist="a1", company="c2",
              year=94,  songs=["s10","s11","s12"]}
            ];
```

Modelling example: CD Register (functions)

Functions on CD registers:

```
fun titles(_, []: cdRegister)                  = []
  | titles(a, {artist, title, ...}::cdreg) =
             if a=artist then title::titles(a, cdreg)
             else titles(a, cdreg);

- titles("a1", cdreg);
> val it = ["t1", "t3"] : string list
```

Modelling example: Cash Register

We make a program for a simple cash register.

- A *data register* associates the *name* and *price* of the article to each valid *article code*.
- A *purchase* is a sequence of *items*, each *item* describes the purchase of a *number of pieces* of a specific *article*
- Construct a program which makes a *bill* of a purchase. Each *item* on the *bill* must contain the *name* of the *article*, the *number of pieces* and the total price. Also, the *bill* must contain the grand total for the entire *purchase*.

Modelling example: Cash Register (decl.)

```
type articleCode = string
type articleName = string
type noPieces    = int
type price       = int
type register = (articleCode *
                 (articleName * price)) list
type item    = noPieces * articleCode
type purchase= item list
type info    = noPieces * articleName * price
type infoseq = info list
type bill    = infoseq * price
exception FindArticle
makeBill: purchase * register -> bill
```

Modelling example: Cash Register (functions)

```
fun findArticle(ac, (ac',adesc)::reg) =
          if ac=ac' then adesc
          else findArticle(ac,reg)
  | findArticle _ =
          raise FindArticle;

fun makeBill([], _)                = ([],0)
  | makeBill((np,ac)::pur, reg) =
        let val (aname,aprice) = findArticle(ac,reg)
            val tprice         = np*aprice
            val (billtl,sumtl) = makeBill(pur,reg)
        in ((np,aname,tprice)::billtl, tprice+sumtl)
        end;
```

Modelling example: Cash Register (testing)

```
- val register =
   [("a1",("cheese",25)),
    ("a2",("herring",4)),
    ("a3",("soft drink",5))
   ];
> val register =
    [("a1", ("cheese", 25)), ("a2", ("herring", 4)),
     ("a3", ("soft drink", 5))]
     : (string * (string * int)) list
- val pur = [(3,"a2"),(1,"a1")];
> val pur = [(3, "a2"), (1, "a1")]
          : (int * string) list
- makeBill(pur,register);
> val it = ([(3, "herring", 12),
             (1, "cheese", 25)], 37)
             : (int * string * int) list * int
```

Exercises - Modelling

- 6.2 - Dating Bureau
- 6.3 - Map-colouring

Next Friday:
One-day project: Piecewise linear curves
In the databar from 8:15 to 12
Prepare by reading the problem formulation on the course homepage: www.imm.dtu.dk/courses/02153