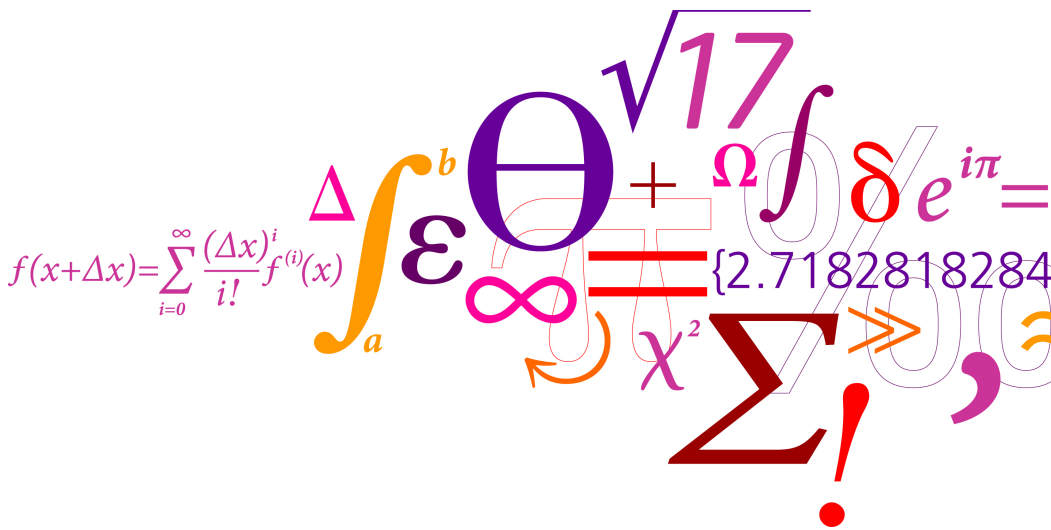
02157 Functional Programming

Lecture 1: Introduction and Getting Started

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

WELCOME to
02157 Functional Programming

Teacher: Michael R. Hansen, DTU Compute mire@dtu.dk

Teaching assistants:

Kristian Hedemark Krarup
Maliina Bolethe Skytte Hammeken
Matias Daugaard Holst-Christensen
Per Lange Laursen
Van Anh Thi Trinh

Homepage: www.compute.dtu.dk/courses/02157

The Functional Setting

A program *f* is a function

$$f : Argument \to Result$$

- *f* takes one argument and produces one result
- arguments and results can be composite values
- computation is governed by function application

For example:

```
insert(4, [1; 3; 7; 9]) = [1; 3; 4; 7; 9]
```

- the argument is a pair: (4, [1; 3; 7; 9])
- the result is a list: [1; 3; 4; 7; 9]
- the computation is guided by repeated function application

```
      insert(4, [1; 3; 7; 9])
⇝     1::insert(4, [3; 7; 9])
⇝     1::3::insert(4, [7; 9])
⇝     1::3::4::[7;9]
=     [1;3;4;7;9]
```

A simple problem solving technique

Solve a complex problem by

- partitioning it into smaller well-defined parts
- compose the parts to solve the original problem.

The main goal:

A program is constructed by
combining simple well-understood pieces

- A general technique that is natural in functional programming.

Insertion in an ordered list is a well-understood program.

- Can you use this in the construction of program for sorting a list?

A typed functional programming language

Supports

- Composite values like lists and trees
- Functions as ”first-class citizens”
- Recursive functions
- Patterns
- A strong polymorphic type system
- Type inference

A small collection of powerful concepts used to explain the meaning of a program

- binding
- environment
- evaluation of expressions

A value-oriented (declarative) programming approach where you focus on *what* properties a solution should rather than on the individual computation steps.

A typed functional programming language

does not support

- assignable variables, assignments
  `a[i] = a[i] + 1, x++, ...`
- imperative control statements `while i<10 do ...`
  `if b then a[i]=a[i]+1,...`
- object-oriented state-changing features `child.age = 5`
- ...

In imperative and object-oriented programming approaches focus is on *how* a solution is obtained in terms of a sequence of state changing operations.

About functional programming

Some advantages of functional programming:

- Supports fast development based on abstract concepts
  more advanced applications are within reach
- Supplements modelling and problem solving techniques
- Supports parallel execution on multi-core platforms

F# is as efficient as C#

Testimonials, quotes and real-world applications of FP languages:

- `http://fsharp.org`
- `homepages.inf.ed.ac.uk/wadler/realworld`

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

Some functional programming background

- Introduction of λ-calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.
- Functional languages (SML, Haskell, OCAML, F#, . . .) have now applications far away from their origin: Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .
- Declarative aspects are now sneaking into ”main stream languages”
- Peter Sestoft: Programming Language concepts, Springer 2012. Uses F# as a meta language.

Practical Matters (I)

- Textbook: Functional Programming using F#
  M.R. Hansen and H. Rischel. Cambridge Univ. Press, 2013.

  `www.imm.dtu.dk/~mrh/FSharpBook`

  Available at DTU's bookstore and library.

- F# (principal designer: Don Syme) is an open-source functional language. F# is integrated in the Visual Studio platform and with access to all features in the .NET program library. The language is also supported on Linux, MAC, . . . systems

- Look at `http://fsharp.org` concerning installations for your own laptops (Windows, Linux, Mac, Android, iPhone, . . .).
  Suggestions: Windows: Visual Studio and for Mac and Linux: Visual Studio Code

- Lectures: Friday 8.15 - 10:00 in Building 341, Auditorium 21

- Exercises classes: Friday 10:00 - 12:00 in Building 341
  Rooms 003, 015 and 019

Practical Matters: Mandatory assignments

- Four mandatory assignments. Dates will be announced soon
- Three must be approved in order to participate in the exam.
- They do not have a role in the final grade.
- Group af size 1, 2 and 3 are allowed.

Approvals of mandatory assignments from earlier years
DO NOT apply this year

Course context

- Prerequisites for 02157:
  - Programming in an imperative/object-oriented language
  - Discrete mathematics (previously or at this semester)

- 02157 is a prerequisite for 02141 Computer Science Modelling

- 02141 is a prerequisite for 02257 Applied Functional Prog.

- 02257 Applied Functional Programming January course

  – efficient use of functional programming in connection with activities at the M.Sc. programmes and in "practical applications".

  One project every week. Last year's projects:

  - Computer science applications.
    — Compiler for a rich imperative programming language.
  - "Practical applications".
    — A game with a reactive graphical user-interface
  - Functional pearls.
    — Layout of trees.

Overview

Part 1 Getting Started:

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Main ingredients of F#

Part 2 Lists:

- Lists: values and constructors
- Recursions following the structure of lists
- Polymorphism

A value-oriented approach

GOAL: By the end of the day you have constructed succinct, elegant and understandable F# programs.

The Interactive Environment

```
2*3 + 4;;
val it : int = 10
```

⇐ Input to the F# system

⇐ Answer from the F# system

- The *keyword* `val` indicates a value is computed
- The *integer* 10 is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The notion *binding* explains which entities are named by identifiers.

`it` $\mapsto$ 10 reads: "`it` is bound to 10"

Value Declarations

A value declaration has the form: `let` *identifier* = *expression*

```
let price = 25 * 5;;
```
⇐ A declaration as input

```
val price : int = 125
```
⇐ Answer from the F# system

The effect of a declaration is a binding: `price` $\mapsto$ 125

Bound identifiers can be used in expressions and declarations, e.g.

```
let newPrice = 2*price;;
val newPrice : int = 250

newPrice > 500;;
val it : bool = false
```

A collection of bindings

$$\begin{bmatrix} \texttt{price} & \mapsto & 125 \\ \texttt{newPrice} & \mapsto & 250 \\ \texttt{it} & \mapsto & \text{false} \end{bmatrix}$$

is called an environment

Function Declarations 1: `let f x = e`

- *x* is called the *formal parameter*
- the defining expression *e* is called the *body* of the declaration

Declaration of the circle area function:

```
let circleArea r = System.Math.PI * r * r;;
```

- `System.Math` is a program library
- `PI` is an identifier (with type `float`) for π in `System.Math`

The type is automatically inferred in the answer:

```
val circleArea : float -> float
```

Applications of the function:

```
circleArea 1.0;; (* this is a comment *)
val it : float = 3.141592654

circleArea(3.2);; // A comment: optional brackets
val it : float = 32.16990877
```

`1.0` and `3.2` are also called *actual parameters*

Recursion. Example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$\begin{aligned} 0! &= 1 && (i) \\ n! &= n \cdot (n-1)!, \quad \text{for } n > 0 && (ii) \end{aligned}$$

- $n!$ is defined recursively in terms of $(n-1)!$ when $n > 0$

Computation:

$$\begin{aligned} & 3! \\ = \; & 3 \cdot (3-1)! && (ii) \\ = \; & 3 \cdot 2 \cdot (2-1)! && (ii) \\ = \; & 3 \cdot 2 \cdot 1 \cdot (1-1)! && (ii) \\ = \; & 3 \cdot 2 \cdot 1 \cdot 1 && (i) \\ = \; & 6 \end{aligned}$$

Function Declarations 2: `let` *rec f x* = *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =
      if n=0 then 1                    (*  i *)
      else n * fact(n-1);;             (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lll}
 & \texttt{fact(3)} & \\
\rightsquigarrow & \texttt{3 * fact(3 - 1)} & (ii) \quad [\texttt{n} \mapsto 3] \\
\rightsquigarrow & \texttt{3 * 2 * fact(2 - 1)} & (ii) \quad [\texttt{n} \mapsto 2] \\
\rightsquigarrow & \texttt{3 * 2 * 1 * fact(1 - 1)} & (ii) \quad [\texttt{n} \mapsto 1] \\
\rightsquigarrow & \texttt{3 * 2 * 1 * 1} & (i) \quad [\texttt{n} \mapsto 0] \\
\rightsquigarrow & \texttt{6} &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

- An environment is used to bind the formal parameter *n* to actual parameters $3, 2, 1, 0$ during evaluation

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `&&` e_2	"conjunction $e_1 \wedge e_2$"
e_1 `||` e_2	"disjunction $e_1 \vee e_2$"

— are lazily evaluated, e.g.

```
1<2 || 5/0 = 1
⇝ true
```

Precedence: `&&` has higher than `||`

If-then-else expressions

Form:

$$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

Evaluation rules:

$$\begin{array}{lcl} \texttt{if true then}\ e_1\ \texttt{else}\ e_2 & \leadsto & e_1 \\ \texttt{if false then}\ e_1\ \texttt{else}\ e_2 & \leadsto & e_2 \end{array}$$

Notice:

- Both then and else branches must be present (because it is an expression that gives a value)
- either e_1 or e_2 is evaluated provided that (but never both) provided that b terminates.

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`String.length`	`string -> int`	length of string
`+`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`string`	`obj -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"+"de";
> val it = "abcde": string
```

```
- String.length("abc"^"def");
> val it = 6 : int

- string(6+18);
> val it = "24": string
```

Patterns

A pattern is composed from identifiers, constants and the wildcard pattern _ using constructors (considered soon)

Examples of patterns are: `3.1`, `true`, `n`, `x`, `5`, `_`

- A pattern may match a value, and if so it results in an environment with bindings for every identifier in the pattern.
- The wildcard pattern `_` matches any value (resulting in no binding)

Examples:

- The value `3.1` matches pattern `x` resulting in environment: $[x \mapsto 3.1]$
- The value `true` matches pattern `true` resulting in the empty environment $[\,]$

Match expressions

A match expression e_m has the following form:

$$\begin{array}{l}\texttt{match}\ e\ \texttt{with}\\ \quad \mid pat_1 \quad \rightarrow \quad e_1\\ \qquad\qquad \vdots\\ \quad \mid pat_n \quad \rightarrow \quad e_n\end{array}$$

A match expression e_m is evaluated as follows:

If

- v is the value of e and
- pat_i is the first matching pattern for v and env is the environment obtained from the pattern matching,

then

$$e_m \rightsquigarrow (e_i, env)$$

Match expressions: An example

Function declaration:

```
let rec fact n =
    match n with
    | 0 -> 1                    (*  i *)
    | n -> n * fact(n-1)        (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lll}
 & \texttt{fact}(3) & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) \quad [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) \quad [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) \quad [\texttt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) \quad [\texttt{n} \mapsto 0] \\
\rightsquigarrow & 6 &
\end{array}
$$

A match with a when clause and an exception:

```
let rec fact n =
    match n with
    | 0           -> 1
    | n when n>0  -> n * fact(n-1)
    | _           -> failwith "Negative argument"
```

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$x^0 = 1 \quad (1)$$
$$x^n = x \cdot x^{n-1}, \quad \text{for } n > 0 \quad (2)$$

Function declaration:

```
let rec power(x,n) =
   match (x,n) with
   | (_,0) -> 1.0                    (* 1 *)
   | (x,n) -> x * power(x,n-1)       (* 2 *)
```

Patterns:

- `(_, 0)` matches any pair of the form $(u, 0)$.
- `(x, n)` matches any pair (u, i) yielding the bindings

$$\texttt{x} \mapsto u, \texttt{n} \mapsto i$$

Evaluation. Example: `power(4.0, 2)`

Function declaration:

```
let rec power(x,n) =
    match (x,n) with
    | (_,0) -> 1.0                    (* 1 *)
    | (x,n) -> x * power(x,n-1)       (* 2 *)
```

Evaluation:

$$
\begin{array}{lll}
 & \texttt{power}(4.0, 2) & \\
\rightsquigarrow & 4.0 * \texttt{power}(4.0, 2-1) & \text{Clause 2, } [\texttt{x} \mapsto 4.0, \texttt{n} \mapsto 2] \\
\rightsquigarrow & 4.0 * \texttt{power}(4.0, 1) & \\
\rightsquigarrow & 4.0 * (4.0 * \texttt{power}(4.0, 1-1)) & \text{Clause 2, } [\texttt{x} \mapsto 4.0, \texttt{n} \mapsto 1] \\
\rightsquigarrow & 4.0 * (4.0 * \texttt{power}(4.0, 0)) & \\
\rightsquigarrow & 4.0 * (4.0 * 1) & \text{Clause 1} \\
\rightsquigarrow & 16.0 &
\end{array}
$$

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Floats	`float`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$
then $(e_1, e_2) : \tau_1 * \tau_2$ pair (tuple) type constructor

Functions:

if $f : \tau_1$ -> τ_2 and $a : \tau_1$ function type constructor
then $f(a) : \tau_2$

Examples:

```
(4.0, 2):  float*int
power:  float*int -> float
power(4.0, 2):  float
```

`*` has higher precedence that `->`

Type inference: `power`

```
let rec power (x,n) =
    match (x,n) with
    | (_,0) -> 1.0                    (* 1 *)
    | (x,n) -> x * power(x,n-1)       (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):float`, because τ_3 = `float`.
- multiplication can have

  `int*int -> int` or `float*float -> float`

  as types, but no "mixture" of `int` and `float`
- Therefore `x:float` and τ_1=`float`.

The F# system determines the type `float*int -> float`

Functions expressions

We have in previous examples exploited pattern matching in match expressions e_m:

$$\begin{array}{l}\texttt{match}\ e\ \texttt{with}\\ \quad \mid pat_1 \quad \rightarrow \quad e_1\\ \qquad\qquad \vdots\\ \quad \mid pat_n \quad \rightarrow \quad e_n\end{array}$$

A function expression e_f has a similar pattern matching feature:

$$\begin{array}{l}\texttt{function}\\ \quad \mid pat_1 \quad \rightarrow \quad e_1\\ \qquad\qquad \vdots\\ \quad \mid pat_n \quad \rightarrow \quad e_n\end{array}$$

- A function expression e_f denotes an anonymous function
- An application $e_f\ e$ is equivalent to the match expression e_m

Anonymous functions: Two examples

An anonymous function computing the number of days in a month:

```
function
| 2  -> 28   // February
| 4  -> 30   // April
| 6  -> 30   // June
| 9  -> 30   // September
| 11 -> 30   // November
| _  -> 31;; // All other months
 val it : int -> int = <fun:clo@17-2>

it 2;;
val it : int = 28
```

An anonymous circle-area function:

```
function r -> System.Math.PI * r * r
```

Alternative declarations of the power function:

```
let rec power = function
              | (_,0) -> 1.0
              | (x,n) -> x * power(x,n-1);;
```

```
let rec power a = match a with
                  | (_,0) -> 1.0
                  | (x,n) -> x * power(x,n-1);;
```

```
let rec power(x,n) = match n with
                     | 0  -> 1.0
                     | n' -> x * power(x,n'-1);;
```

A curried version:

what is the type?

```
let rec power x = function
   | 0           -> 1.0
   | n when n<0  -> failwith "power: negative exponent"
   | n           -> x * power x (n-1);;
```

Summary

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Breath first round through many concepts aiming at program construction from the first day.

We will go deeper into each of the concepts later in the course.

Overview of Part 2: Lists

- Lists: values and constructors
- Recursions following the structure of lists
- Polymorphism

The purpose of this lecture is to give you an (as short as possible) introduction to lists, so that you can solve a problem which can illustrate some of F#'s high-level features.

This part is *not* intended as a comprehensive presentation on lists, and we will return to the topic again later.

Lists

A list is a finite sequence of elements having the same type:

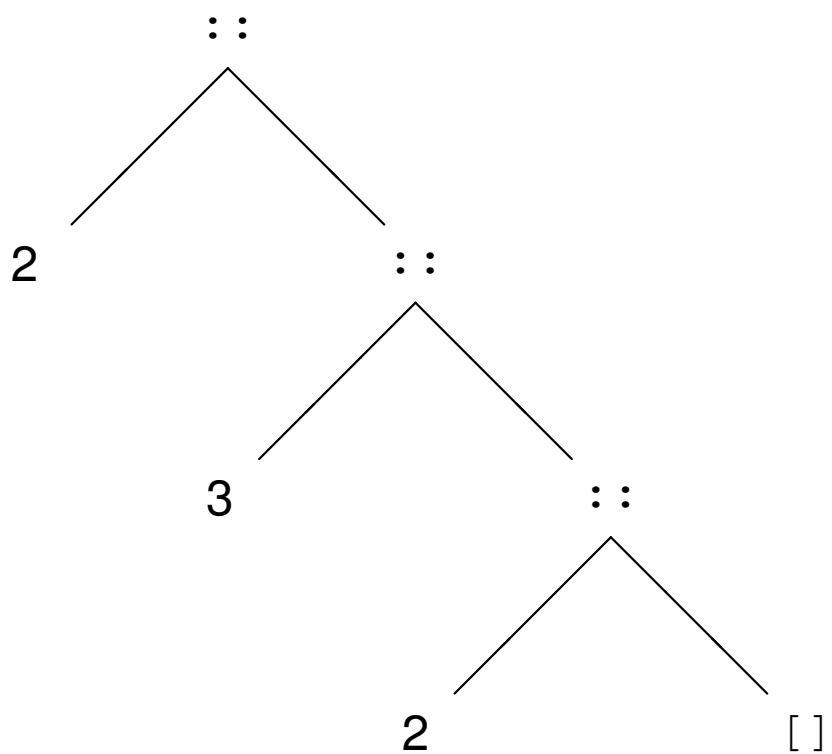
$[v_1; \ldots; v_n]$ ($[\,]$ is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]

[(1,true); (3,true)];;
val it : (int * bool) list = [(1, true); (3, true)]

[[]; [1]; [1;2]];;
val it : int list list = [[]; [1]; [1; 2]]
```

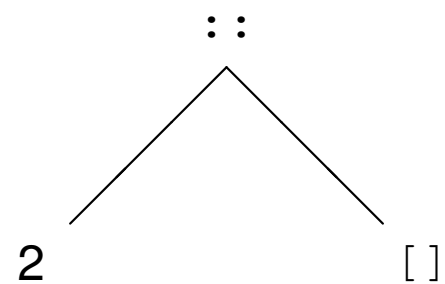
Trees for lists

A non-empty list $[x_1, x_2, \ldots, x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2, \ldots, x_n]$



Graph for `[2,3,2]`



Graph for `[2]`

Recursion on lists – a simple example

$$\texttt{suml}\ [x_1\texttt{;}\,x_2\texttt{;}\,\ldots\texttt{;}\,x_n] = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n = x_1 + \sum_{i=2}^{n} x_i$$

Constructors are used in list patterns

```
let rec suml xs =
    match xs with
    | []      -> 0
    | x::tail -> x + suml tail;;
val suml : int list -> int
```

```
  suml [1;2]
⇝ 1 + suml [2]          (x ↦ 1 and tail ↦ [2])
⇝ 1 + (2 + suml [])     (x ↦ 2 and tail ↦ [])
⇝ 1 + (2 + 0)           (the pattern [] matches the value [])
⇝ 1 + 2
⇝ 3
```

Recursion follows the structure of lists

The Polymorphic Append function

The built-in function append `@` joins two lists:

$$[x_1; x_2; \ldots; x_m] \texttt{ @ } [y_1; y_2; \ldots; y_n] = [x_1; x_2; \ldots; x_m; y_1; y_2; \ldots; y_n]$$

It has a polymorphic type

```
val (@) : 'a list -> 'a list -> 'a list
```

- `'a` is a *type variable*

Append has many forms

`'a = int`: Appending integer lists

```
[1;2] @ [3;4];;
val it : int list = [1;2;3;4]
```

`'a = int list`: Appending lists of integer list

```
[[1];[2;3]] @ [[4]];;
val it : int list list = [[1]; [2; 3]; [4]]
```

Another polymorphic list function

The function remove(y,xs) gives the list obtained from xs by deleting every occurrence of y, e.g. $\texttt{remove}(2, [1;2;0;2;7]) = [1;0;7]$.

Recursion is following the structure of the list:

```
let rec remove(y,xs) =
   match xs with
   | []                -> []
   | x::tail when x=y  -> remove(y,tail)
   | x::tail           -> x::remove(y,tail);;
```

List elements can be of any type that supports equality

```
remove : 'a * 'a list -> 'a list when'a : equality
```

For example

```
remove("a", [""; "a"; "ab"; "a"; "bc"]);;
val it : string list = [""; "ab"; "bc"]
```

Exploiting structured patterns: the isPrefix function

The function `isPrefix`(*xs*, *ys*) tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\texttt{isPrefix}([1;2;3],[1;2;3;8;9]) = \text{true}$$
$$\texttt{isPrefix}([1;2;3],[1;2;8;3;9]) = \text{false}$$

The function is declared as follows:

```
let rec isPrefix(xs, ys) =
match (xs,ys) with
| ([],_)                -> true
| (_,[])                -> false
| (x::xtail,y::ytail) -> x=y && isPrefix(xtail, ytail);;

isPrefix([1;2;3], [1;2]);;
val it : bool = false
```

A each clause expresses succinctly a natural property:

- The empty list is a prefix of any list
- A non-empty list is not a prefix of the empty list
- A non-empty list (...) is a prefix of another non-empty list (...) if ...

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)
- Patterns
- Recursion on the structure of lists
- Constructors used in patterns to decompose structured values
- Constructors used in expressions to compose structured values

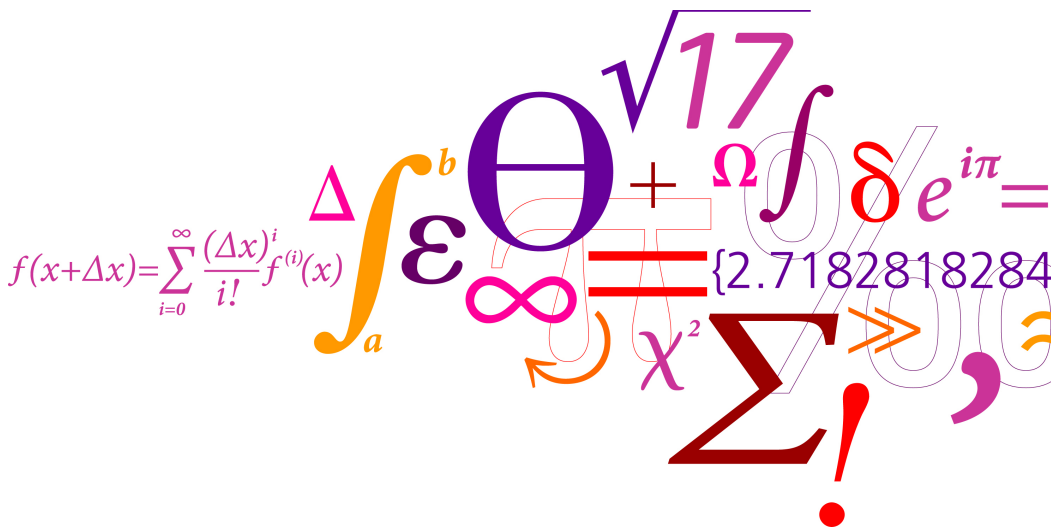
Blackboard exercises

- `memberOf`(x, ys) is true iff x occurs in the list ys
- `insert`(x, ys) is the *ordered list* obtained from the *ordered list* ys by insertion of x
- `sort`(xs) gives a ordered version of xs

02157 Functional Programming

Lecture 2: Functions, Types and Lists

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Outline

- Functions as ”first-class citizens”
- Types, polymorphism and type inference
  - type constraints (equality and comparison)
  - type declarations
- Lists
- Selected language constructs

Goal: By the end of the day you are acquainted with a major part of the F# language.

Functions as ”first-class citizens”

- functions can be passed as arguments to functions
- functions can be returned as values of functions

like any other kind of value.

There is nothing special about functions in functional languages

A function that takes a function as argument or produces a function as result is also called a higher-order function.

Higher-order functions are useful

- succinct code
- highly parametrized programs
- Program libraries typically contain many such functions

Anonymous functions

Function expressions with general patterns, e.g.

```
function
 | 2          -> 28  // February
 |4|6|9|11    -> 30  // April, June, September, November
 | _          -> 31  // All other months
;;
```

Simple function expressions, e.g.

```
fun r -> System.Math.PI * r * r ;;
val it : float -> float = <fun:clo@10-1>

it 2.0 ;;
val it : float = 12.56637061
```

Anonymous functions

Simple functions expressions with *currying*

$$\texttt{fun}\ x\ y\ \cdots\ z\ \rightarrow e$$

with the same meaning as

$$\texttt{fun}\ x \rightarrow (\texttt{fun}\ y \rightarrow (\cdots\ (\texttt{fun}\ z \rightarrow e)\cdots))$$

For example: The function below takes an integer as argument and returns a function of type `int -> int` as value:

```
fun x y -> x + x*y;;
val it : int ->  int -> int = <fun:clo@2-1>

let f = it 2;;
val f : ( int -> int)

f 3;;
val it : int = 8
```

Functions are first class citizens:
the argument and the value of a function may be functions

Function declarations

A simple function declaration:

$$\texttt{let}\ f\ x\ =\ e \qquad \text{means} \qquad \texttt{let}\ f\ =\ \texttt{fun}\ x\ \rightarrow e$$

A declaration of a curried function

$$\texttt{let}\ f\ x\ y\ \cdots z\ =\ e$$

has the same meaning as:

$$\texttt{let}\ f\ =\ \texttt{fun}\ x \rightarrow (\texttt{fun}\ y \rightarrow (\cdots\ (\texttt{fun}\ z \rightarrow e)\cdots))$$

For example:

```
let addMult x y = x + x*y;;
val addMult : int -> int -> int

let f = addMult 2;;
val f : (int -> int)

f 3;;
val it : int = 8
```

An example

Suppose that we have a cube with side length s, containing a liquid with density ρ. The weight of the liquid is then given by $\rho \cdot s^3$:

```
let weight ro s = ro * s ** 3.0;;
val weight : float -> float -> float
```

We can make *partial evaluations* to define functions for computing the weight of a cube of either water or methanol:

```
let waterWeight = weight 1000.0;;
val waterWeight : (float -> float)

waterWeight 2.0;;
val it : float = 8000.0

let methanolWeight = weight 786.5 ;;
val methanolWeight : (float -> float)

methanolWeight 2.0;;
val it : float = 6292.0
```

The formula $\rho \cdot s^3$ is represented just once in the program

Infix functions

The prefix version $(\oplus)$ of an infix operator $\oplus$ is a curried function.

For example:

```
(+);;
val it : (int -> int -> int) = <fun:it@1>
```

Arguments can be supplied one by one:

```
let plusThree = (+) 3;;
val plusThree : (int -> int)

plusThree 5;;
val it : int = 8
```

Function composition: $(f \circ g)(x) = f(g(x))$

For example, if $f(y) = y + 3$ and $g(x) = x^2$, then $(f \circ g)(z) = z^2 + 3$.

The infix operator << in F# denotes function composition:

```
let f y = y+3;;

let g x = x*x;;

let h = f << g;;          // h = (f o g)
val h : int -> int

h 4;;                     // h(4) = (f o g)(4)
val it : int = 19
```

Type of (<<) ?

Types and type checking

Purposes:

- Modelling, readability: types are used to indicate the intention behind a program

- Safety, efficiency: ”Well-typed programs do not go wrong”
  Robin Milner
  - Catch errors at compile time
  - Verification of type properties is not needed at runtime

A type checker is an algorithm used at an early phase in the compiler to check whether a program contains type errors.

Fundamental type-checking problem

All non-trivial semantic properties of programs are undecidable

Rice's theorem

Examples: *p* terminates on all its input

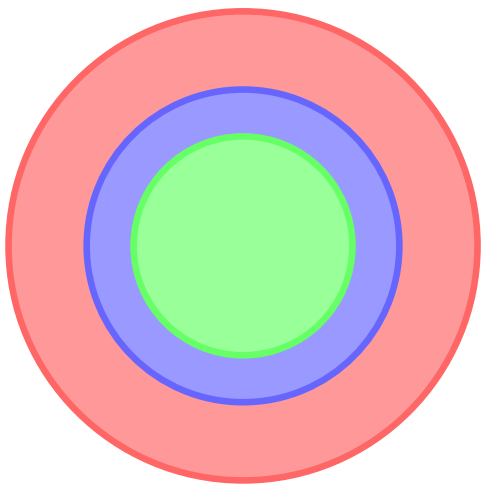
Cannot be checked for programs *p* belonging to a Turing-powerful language

Consequence: A type-checking algorithm provides an approximation:

ill-typed, bad programs

ill-typed, good programs

well-typed, good programs



Type inference

The type system of F# allows for polymorphic types, that is, types with many forms. Polymorphic types are expressed using type variables 'a, 'b, 'c, ...

The *most general type* or *principal type* is inferred by the system.

Examples:

```
let id x = x
val id : 'a -> 'a

let pair x y = (x,y)
val pair : 'a -> 'b -> 'a * 'b
```

The inferred types are most general in the sense that all other types for id and pair are instances of the inferred types.

applications of the functions?

By the type of a function, we (usually) mean the most general type

Remark: identity function id is a built-in function

A simple list recursion

List.append is a function from the `List` library:

- `List.append` $[x_0; \ldots; x_{n-1}]\,[y_0; \ldots; x_{m-1}] =$
  $[x_0; \ldots; x_{n-1}; y_0; \ldots; x_{m-1}]$

There is a convenient infix notation for `List.append` *xs ys* in F#:

xs `@` *ys*

The declaration of `(@)` *xs ys* follows the structure of *xs*:

```
let rec (@) xs ys =
   match xs with
   | []       -> ys
   | x::xtail -> x::(xtail @ ys);;
val ( @ ) : xs:'a list -> ys:'a list -> 'a list

[["a"]; ["ab";"abc"; ""]; []] @ [["x"]; ["xy"; "xyz"]];;
val it : string list list =
    [["a"]; ["ab"; "abc"; ""]; []; ["x"]; ["xy"; "xyz"]]
```

Polymorphic type inference – informally

Given a declaration, for example,

```
let rec (@) xs ys =
    match xs with
    | []        -> ys                          (* C 1 *)
    | x::xtail -> x::(xtail @ ys);;            (* C 2 *)
```

- Guess types for the arguments of `(@)`: *xs* :′ *u* and *ys* :′ *v*
- Add type constraints based on the body of the declaration:
  1. `[]`: 'a list and 'u = 'a list, where 'a is a fresh type variable — C 1
  2. x: 'a, xtail: 'a list, (xtail `@` ys): 'a list, x::(xtail `@` ys): 'a list — C 2
     exploiting the type og `::`
  3. ys : 'a list, 'v = 'a list — C 1,2
     ys must have the same type as x::(xtail `@` ys)

Every sub-expression is now consistently typed.

The most general type or principle type of (`@`) is:

```
'a list -> 'a list -> 'a list
```

- First inference algorithm for ML DamasMilner82
- A nice introduction and F# implementation: Sestoft12

Basic Types: equality and comparison

Equality and comparison are defined for the basic types of F#, including integers, floats, booleans, characters and strings.

Examples:

```
true < false;;
val it : bool = false

'a' < 'A';;
val it : bool = false

"a" < "ab";;
val it : bool = true
```

Composite Types: equality and comparison

Equality and comparison carry over to composite types
as long as function types are not involved:

Equality is defined structurally on values with the same type:

```
(1, true, 7.4) = (2-1, not false, 8.0 - 0.6);;
val it : bool = true

[[1;2]; [3;4;5]] = [[1..2]; [3..5]];;
val it : bool = true
```

Comparison is typically defined using lexicographical ordering:

```
[1; 2; 3] < [1; 4];;
val it : bool = true

(2, [1; 2; 3]) > (2, [1;4]);;
val it : bool = false
```

Polymorphic types: equality and comparison constraints (I)

Polymorphic types may be accompanied with equality and comparison constraints like:

- when 'a : comparison
- when 'b : equality

For example, there is a built-in function:

$$\texttt{compare}\ x\ y \;=\; \begin{cases} > 0 & \text{if } x > y \\ 0 & \text{if } x = y \\ < 0 & \text{if } x < y \end{cases}$$

with the type:

```
'a -> 'a -> int  when 'a : comparison
```

For example:

```
compare (2, [1; 2; 3]) (2, [1;4]);;
val it : int = -1
```

Polymorphic types: equality and comparison constraints (II)

The built-in function List.contains can be declared as follows:

```
let rec contains x =
    function
    | []    -> false
    | y::ys -> x=y || contains x ys
contains: 'a -> 'a list -> bool when 'a : equality

contains [3;4] [[1..2]; [3..5]];;
val it : bool = false
```

Notice:

- The equality constraint in the type
- Lazy (short-circuit) evaluation of $e_1||e_2$ causes termination as soon as an element y equal to x is found
- Yet an evaluation following the structure of lists

Let-expressions

A let-expression e_l has the (verbose) form

```
let x = e1 in e2
```

or the following short form exploiting indentation:

```
let x = e1
e2
```

The expression provides a local definition for x in e2.

A let-expression e_l is evaluated in an environment *env* as follows:

If

1. $v1$ is the value obtained by evaluating $e1$ in *env*,
2. env' is the environment obtained by adding the binding $x \mapsto v$ to *env* and
3. $v2$ is the value obtained by evaluating $e2$ in env'

then

$$(\texttt{let}\ x = e1\ \texttt{in}\ e2, env) \rightsquigarrow (v2, env)$$

Let-expression – an example

Examples

```
let g x = let a = 6
          let b = x + a
          x + b;;
val g : int -> int

g 1;;
val it : int = 8
```

Note: a and b are not visible outside of g

Tuples

An ordered collection of n values $(v_1, v_2, \ldots, v_n)$ is called an n-tuple

Examples

`(3, false);` `val it = (3, false) : int * bool`	2-tuples (pairs)
`(1, 2, ("ab",true));` `val it = (1, 2, ("ab", true)) : ?`	3-tuples (triples)

Equality defined componentwise, ordering lexicographically

```
(1, 2.0, true) = (2-1, 2.0*1.0, 1<2);;
val it = true : bool

compare (1, 2.0, true) (2-1, 3.0, false);;
val it : int = -1
```

provided = is defined on components

Tuple patterns

Extract components of tuples

```
let ((x,_),(_,y,_)) = ((1,true),("a","b",false));;
val x :  int = 1
val y :  string = "b"
```

Pattern matching yields bindings

Restriction

```
let (x,x) = (1,1);;
...
 ...  ERROR ...  'x' is bound twice in this pattern
```

Restriction can be circumvented using `when` clauses, for example:

```
let f = function
        | (x,y) when x=y -> x
        | (x,y)          -> x+y
```

Pattern matching on results of recursive calls

$$\texttt{sumProd}\ [x_0;x_1;\ldots;x_{n-1}] = (x_0 + x_1 + \ldots + x_{n-1}\ ,\ x_0 * x_1 * \ldots * x_{n-1})$$
$$\texttt{sumProd}\ [] = (0,1)$$

The declaration is based on the recursion formula:

$$\texttt{sumProd}\ [x_0;x_1;\ldots;x_{n-1}] = (x_0 + \texttt{rSum}, x_0 * \texttt{rProd})$$

where $(\texttt{rSum},\texttt{rProd}) = \texttt{sumProd}\ [x_1;\ldots;x_{n-1}]$

This gives the declaration:

```
let rec sumProd =
    function
    | []      -> (0,1)
    | x::rest -> let (rSum,rProd) = sumProd rest
                 (x+rSum,x*rProd);;
val sumProd : int list -> int * int

sumProd [2;5];;
val it : int * int = (7, 10)
```

A blackboard exercise

A function from the `List` library:

- `List.unzip([(`x_0, y_0`);(`x_1, y_1`);...;(`x_{n-1}, y_{n-1}`)]`
  `=([`x_0`;`x_1`;...;`x_{n-1}`],[`y_0`;`y_1`;...;`y_{n-1}`])`

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`let square x = x * x`	`int -> int`	Default

A squaring function on floats: `square:  float -> float`

Declaration	
`let square(x:float) = x * x`	Type the argument
`let square x:float = x * x`	Type the result
`let square x = x * x:  float`	Type expression for the result
`let square x = x:float * x`	Type a variable

You can mix these possibilities

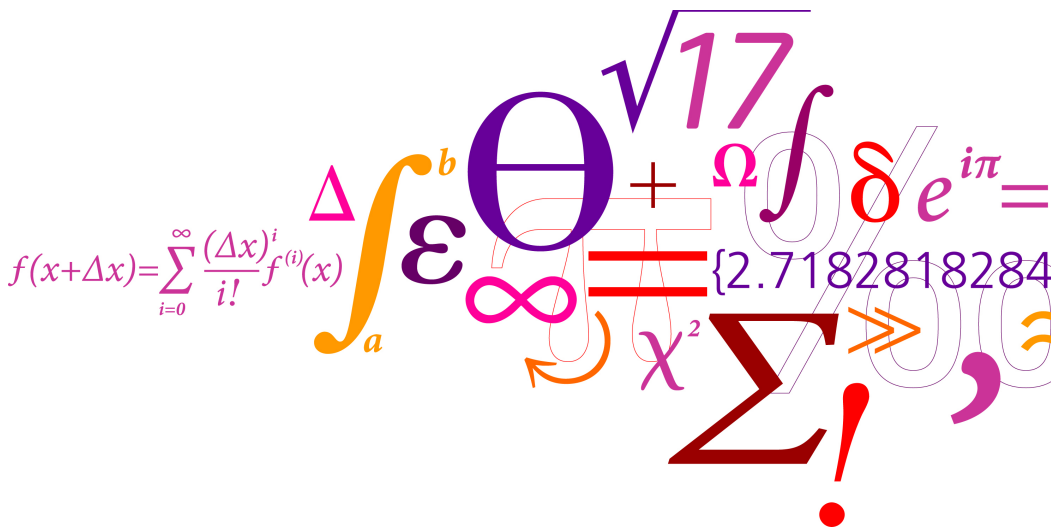
Summary

- Functions as ”first-class citizens”
- Types, polymorphism and type inference
  - type constraints (equality and comparison)
  - type declarations
- Lists
- Selected language constructs

02157 Functional Programming

Lecture 3: Programming as a model-based activity

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- RECAP
  - Higher-order functions (lists)
  - Type inference
- Syntax, semantics and pragmatics (briefly)
  - Value polymorphism restriction — relates to syntax and semantics
  - Simplification of programs — relates to pragmatics
- Programming as a modelling activity — relates to pragmatics
  - Type declarations (type abbreviations)
  - Cash register
  - Map coloring

Exploiting functional decomposition

Example: takeWhile

a predicate is an argument

A predicate is a truth-valued function.

- `takeWhile` : $('a \to \texttt{bool}) \to 'a\ \texttt{list} \to 'a\ \texttt{list}$, where
  `takeWhile` p `[`x_1`;`x_2`; ...;`x_n`]` = `[`x_1`;`x_2`; ...;`x_k`]`

where $p\ x_i = \text{true}, 1 \leq i \leq k$ and ($k = n$ or $k < n$ and $p\ x_{k+1} = \text{false}$).

```
let rec takeWhile p =
    function
    | x::xs when p x -> x::takeWhile p xs
    | _              -> []

let xs = [3; 5; 1; 6];;

takeWhile (fun x -> x>1) xs;;
val it : int list = [3; 5]

takeWhile  (fun x -> x%2=1) xs;;
val it : int list = [3; 5; 1]
```

A higher-order list function that takes a predicate as argument

Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.
- The semantics is concerned with the meaning of syntactically correct programs.
- The pragmatics is concerned with practical (adequate) application of language constructs in order to achieve certain objectives.

Syntactical constructs in F#

- Constants: 0, 1.1, true, ...
- Patterns:
  x `_` $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions:
  x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $e_1\, e_2$ $e_1 \oplus e_2$ `let` $p_1 = e_1$ `in` e_2 $e : t$
  `match` e `with` *clauses* `fun` $p_1 \cdots p_n$`->`e `function` *clauses* $\ldots$
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int` `bool` `string` $'a$ $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2 \ldots$

where the construct *clauses* has the form:

```
| p1 -> e1 | ... | pn -> en
```

In addition to that

- precedence and associativity rules, parenthesis around p and e and type correctness

Semantics of F# programs

The semantics can be described by rules: $\frac{\text{List of hypothesis}}{\text{Conclusion}}$

Conclusion and hypotheses are *judgements*: $env \vdash e => v$

- "within environment env, evaluation of expression e terminates and gives value v".

We show a few rules from semantics of micro-ML: Sestoft2012

$$\frac{env(x) = v}{env \vdash x => v} \qquad \frac{env \vdash e_1 => v_1 \quad env \vdash e_2 => v_2 \quad v_1 + v_2 = v}{env \vdash e_1 + e_2 => v}$$

Let-expression: $$\frac{env \vdash e_r => v_r \quad env[x \mapsto v_r] \vdash e_b => v_b}{env \vdash \texttt{let}\ x = e_r\ \texttt{in}\ e_b => v_b}$$

If-then-else:

$$\frac{env \vdash e => false \quad env \vdash e_2 => v_2}{env \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 => v_2} \text{ and } \frac{env \vdash e => true \quad env \vdash e_1 => v_1}{env \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 => v_1}$$

Semantics: Functions and function application

What is the value of the following expression?

```
let a = 1
in let f = fun x -> x+a
   in let a = 2
      in f 3;;
```

The semantics of a function, e.g. `fun` $x->e$ in environment env_{dec}:

(x, e, env_{dec}) is called a closure

.

Application of (non-recursive) functions:

$$\frac{env \vdash e_1 => (x, e, env_{dec}) \quad env \vdash e_2 => v_2 \quad env_{dec}[x \mapsto v_2] \vdash e => v}{env \vdash e_1\, e_2 => v}$$

- the closure for f is $(x, x + a, [a \mapsto 1])$
- f 3 is evaluated in environment $[a \mapsto 2, f \mapsto (x, x + a, [a \mapsto 1])]$
- x+a is evaluated in the environment $[a \mapsto 1, x \mapsto 3]$
- the value of the expression is 4

What if f 3 is changed to f a?

Semantics: observations

- The environment in which a function is declared is a part of its closure F# has static binding
  – the fresh binding for a does not influence the meaning of f

- Just a slight extension is required in order to cope with recursive functions.

- The informal treatment of step-by-step evaluations in this course ($e_1 \rightsquigarrow e_2$) is based on a formal semantics for functional programming languages

- The pure functional programming part of F# consists of a small number of constructs with simple well-defined semantics.

- Syntax and semantics of programming languages is a part of 02141 Computer Science Modelling

Examples on errors with polymorphic functions (1)

```
let empty = [] @ [];;
stdin(3,5): error FS0030: Value restriction.
The value 'empty' has been inferred to have generic type
val empty : '_a list

Either define 'empty' as a simple data term,
make it a function with explicit arguments or,
if you do not intend for it to be generic,
   add a type annotation.

let empty = [];;  // WORKS!
val empty : 'a list
```

Examples on errors with polymorphic functions (2)

```
List.rev [];;
stdin(5,1): error FS0030: Value restriction.
The value 'it' has been inferred to have generic type
val it : '_a list

Either define 'it' as a simple data term,
make it a function with explicit arguments or,
if you do not intend for it to be generic,
   add a type annotation.


List.rev ([]: int list);;  // WORKS!
val it : int list = []
```

- a top-level expression is considered a top-level declaration of `it`:

```
let it = List.rev ([]: int list);;
val it : int list = []
```

Examples on errors with polymorphic functions (3)

```
let takeAll = takeWhile (fun _ -> true);;
stdin(15,5): error FS0030: Value restriction
The value 'takeAll' has been inferred to have generic type
    val takeAll : ('_a list -> '_a list)

Either make the arguments to 'takeAll' explicit or
if you do not intend for it to be generic,
   add a type annotation.
```

```
let takeAll xs = takeWhile (fun _ -> true) xs;;  // WORKS!
val takeAll : xs:'a list -> 'a list
```

- What is the problem solved by this restriction?
- What is the solution to this problem?

An ”imperative” issue with Polymorphism

- A type issue to be addressed in the imperative fragment of F#.

The following hypothetical code does NOT compile:

```
let mutable a = []                    // let a = ref []

let f x = a <- x::a
 val f: 'a -> unit

f 1;;

f "ab";;

a;;
val it :  ???? list = ["ab";1] // a BIG type problem
```

A solution: Value Restriction on Polymorphism

A value expression is an expression that is not reduced further by an evaluation, for example:

```
[]      Some []     fun xs -> takeWhile (fun _ -> true) xs
```

The following are not value expressions:

```
[]@[]      List.rev []      List.map (fun _ -> true)
```

as they can be further evaluated.

Every top-level declarations `let` *id* `=` *e* is subject to the following restriction on *e*:

- All monomorphic expressions are OK,
- all value expressions are OK, even polymorphic ones, and
- at top-level, polymorphic non-value expressions are forbidden

An expression `ref` *e* in a declaration of the form
`let` *id* `= ref` *e* (or equivalently `let mutable` *id* `=` *e*)
is NOT considered a value expression (even when *e* is a constant)

Examples revisited

A polymorphic value expression:

```
let empty = [];;  // WORKS!
val empty : 'a list
```

A monomorphic expression, that is not a value expression:

```
List.rev ([]: int list);;  // WORKS!
val it : int list = []
```

A declaration of a function with an explicit parameter

```
let f xs = takeWhile (fun _ -> true) xs;;  // WORKS!
val f : xs:'a list -> 'a list
```

is an abbreviation for

```
let f = fun xs -> takeWhile (fun _ -> true) xs;;
```

and a closure is a value expression

On “simplifying” programs (1)

Programs with the following flavour are too often observed:

```
let f(x) = match (x) with
           |(a,z) -> if (not(a) = true) then true
                     else if (fst(z) = true) then snd(z)
                          else false;;
```

- What is the type of *f*?
- What is *f* computing?

Some rules of thump:

- Avoid obvious superfluous use of parenthesis
- Simplify Boolean expressions
- A match with just one clause can be avoided
- Use of `fst` and `snd` is avoided using patterns
- `if-then-else` expressions having truth values are avoided using Boolean expressions

On “simplifying” programs (2)

```
let f(x) = match (x) with
           |(a,z) -> if (not(a) = true) then true
                     else if (fst(z) = true) then snd(z)
                          else false;;
```

Avoid obvious superfluous use of parenthesis:

```
let f x = match x with
          |(a,z) -> if not a = true then true
                    else if fst z = true then snd z
                         else false;;
```

Simplify Boolean expressions:

```
let f x = match x with
          |(a,z) -> if not a then true
                    else if fst z then snd z
                         else false;;
```

- `e = true` has the same truth value as `e`

On “simplifying” programs (3)

```
let f x = match x with
          |(a,z) -> if not a then true
                    else if fst z then snd z
                         else false;;
```

Avoid a match with just one clause:

```
let f(a,z) = if not a then true
             else if fst z then snd z else false;;
```

Avoid `fst` and `snd`:

```
let f(a,(b,c)) = if not a then true
                 else if b then c else false;;
```

On “simplifying” programs (4)

- `if-then-else` expressions having truth values are avoided using Boolean expressions

```
let f(a,(b,c)) = if not a then true
                 else if b then c else false;;
```

`if` *p* `then` *q* `else false` is the same as *p* `&&` *q*:

```
let f(a,(b,c)) = if not a then true else b && c;;
```

`if` *p* `then true else` *q* is the same as *p* `||` *q*:

```
let f(a,(b,c)) = not a || b && c;;
```

On “simplifying” programs (5)

```
let f(a,(b,c)) = not a || b && c;;
```

- The type of *f* is `bool*(bool*bool) -> bool`
- $f(a, (b, c))$ is the value of the proposition “*a* implies (*b* and *c*)”

Introducing implication:

```
let (.=>) p q = not p || q;;

let f(a,(b,c)) = a .=> (b && c);;
```

Type declarations (abbreviations)

A declarations of a monomorphic type has the form:

$$\texttt{type}\ T = t$$

where t is a type expression not containing type variables.

A declaration of a polymorphic type has the form:

$$\texttt{type}\ T <' v_0,' v_1, \ldots,' v_n >= t$$

where t is a type expression containing type variables $'v_0,' v_1, \ldots,' v_n$.

Type constraints may be added to the type-parameter list.

Type declarations (abbreviations): Examples

```
type Name = string
type Id = int
type Assoc<'K, 'V  when 'K : equality> = ('K * 'V) list

type Participants = Assoc<Id,Name>
let ex:Participants = [(164255, "Bill") ; (173333,"Eve")]
val ex : Participants = [(164255, "Bill"); (173333, "Eve")]

ex =  ([(0,"")]: (int*string) list) ;;
val it : bool = false

let rec insert k v (ass:Assoc<'K,'V>) = (k,v)::ass;;
val insert : 'K -> 'V -> Assoc<'K,'V>
                      -> ('K * 'V) list when 'K : equality

insert 1 "a" [(0,"")];;
val it : (int * string) list = [(1, "a"); (0, "")
```

- The declared types work as abbreviations

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type ArticleCode = string
type ArticleName = string
type Price       = int
type Register    = (ArticleCode * (ArticleName*Price)) list
type NoPieces    = int
type Item        = NoPieces * ArticleCode
type Purchase    = Item list
type Info        = NoPieces * ArticleName * Price
type Infoseq     = Info list
type Bill        = Infoseq * Price

makeBill: Register -> Purchase -> Bill
```

Is the model adequate?

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Functional decomposition (1)

Type: findArticle: ArticleCode → Register → ArticleName * Price

```
let rec findArticle ac = function
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                    -> findArticle ac reg
    | _                         ->
          failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

An article description is found as follows:

```
findArticle "a2" reg;;
val it : string * int = ("herring", 4)

findArticle "a5" reg;;
System.Exception: a5 is an unknown article code
   at FSI_0016.findArticle[a] ...
```

Note: failwith is a built-in function that raises an exception

Functional decomposition (2)

Type: makeBill: Register → Purchase → Bill

```
let rec makeBill reg = function
    | []              -> ([],0)
    | (np,ac)::pur ->
        let (aname,aprice) = findArticle ac reg
        let tprice         = np*aprice
        let (billtl,sumtl) = makeBill reg pur
        ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

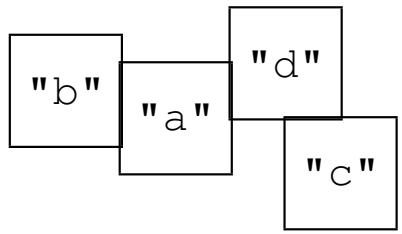
```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                  -> (int * 'a * int) list * int


makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

Example: Map Coloring.

Color a map so that neighbouring countries get different colors



The types for country and map are “straightforward”:

- `type Country = string`

  Symbols: c, c1, c2, c’; Examples: ”a”, ”b”, . . .

- `type Map=(Country*Country) list`

  Symbols: m; Example: val exMap = [(”a”,”b”); (”c”,”d”); (”d”,”a”)]

How many ways could above map be colored?

Abstract models for color and coloring

- `type Color = Country list`

  Symbols: col; Example: [”c”; ”a”]

- `type Coloring = Color list`

  Symbols: cols; Example: [[”c”; ”a”]; [”b”; ”d”]]

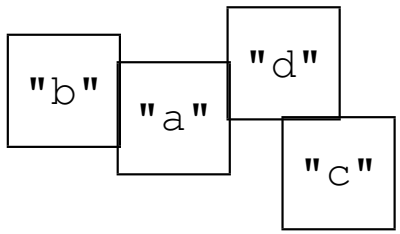
Be conscious about symbols and examples

```
colMap:  Map -> Coloring
```

Meta symbol: Type	***Definition***	***Sample value***
`c:  Country`	`string`	`"a"`
`m:  Map`	`(Country*Country) list`	`[("a","b"),("c","d"),("d","a")]`
`col:  Color`	`Country list`	`["a","c"]`
`cols: Coloring`	`Color list`	`[["a","c"],["b","d"]]`

Figure: A Data model for map coloring problem

Algorithmic idea



Insert repeatedly countries in a coloring.

	country	old coloring	new coloring
1.	`"a"`	`[]`	`[["a"]]`
2.	`"b"`	`[["a"]]`	`[["a"] ; ["b"]]`
3.	`"c"`	`[["a"] ; ["b"]]`	`[["a";"c"] ; ["b"]]`
4.	`"d"`	`[["a";"c"] ; ["b"]]`	`[["a";"c"] ; ["b";"d"]]`

Figure: Algorithmic idea

Functional decomposition (I)

To make things easy

Are two countries neighbours?

areNb: Map → Country → Country → bool

```
let areNb m c1 c2 = isMember (c1,c2) m || isMember (c2,c1) m;;
```

Can a color be extended?

canBeExtBy: Map → Color → Country → bool

```
let rec canBeExtBy m col c =
  match  col with
  | []        -> true
  | c'::col' -> not (areNb m c' c) && canBeExtBy m col' c;;

canBeExtBy exMap ["c"] "a";;
val it : bool = true

canBeExtBy exMap ["a"; "c"] "b";;
val it : bool = false
```

Functional composition (I)

Combining functions make things easy

Extend a coloring by a country:

extColoring: Map → Coloring → Country → Coloring

```
Examples:
  extColoring exMap [] "a"       =  [["a"]]
  extColoring exMap [["b"]] "a"  =  [["b"] ; ["a"]]
  extColoring exMap [["c"]] "a"  =  [["a"; "c"]]
```

```
let rec extColoring m cols c =
    match cols with
    | []          -> [[c]]
    | col::cols' -> if canBeExtBy m col c
                    then (c::col)::cols'
                    else col::extColoring m cols' c;;
```

Function types, consistent use of symbols, and examples make program easy to comprehend

Functional decomposition (II)

To color a neighbour relation:

- Get a list of countries from the neighbour relation.
- Color these countries

Get a list of countries without duplicates:

```
let addElem x ys = if isMember x ys then ys else x::ys;;

let rec countries = function
    | []           -> []
    | (c1,c2)::m -> addElem c1 (addElem c2 (countries m));;
```

Color a country list:

```
let rec colCntrs m = function
    | []    -> []
    | c::cs -> extColoring m (colCntrs m cs) c;;
```

Functional composition (III)

The problem can now be solved by
combining well-understood pieces

Create a coloring from a neighbour relation:

colMap: Map $\rightarrow$ Coloring

```
let colMap m = colCntrs m (countries m);;

colMap exMap;;
val it : string list list = [["c"; "a"]; ["b"; "d"]]
```

On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.
- Examples may help understanding the problem and its solution.
- Functional paradigm is powerful.

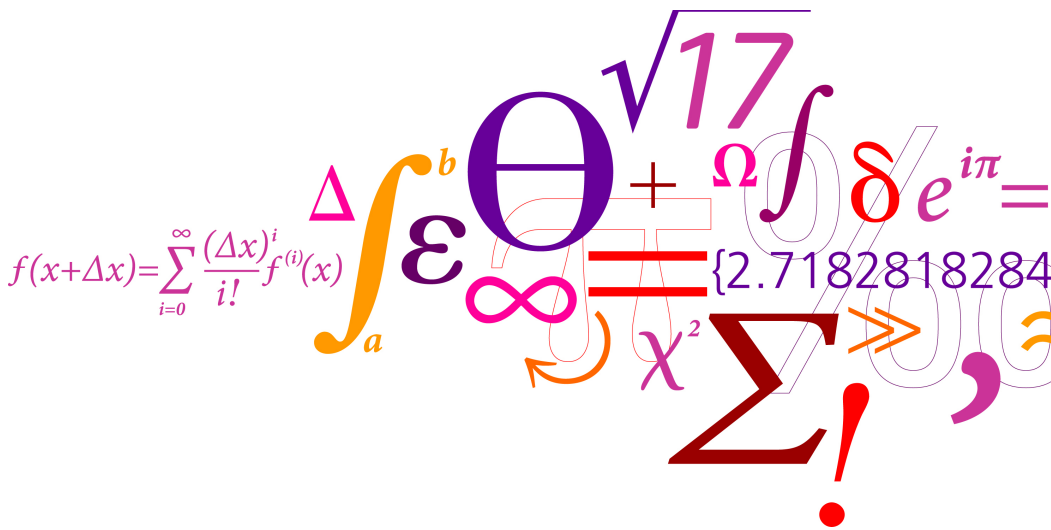
  Problem solving by combination of well-understood pieces

These points are not programming language specific

02157 Functional Programming

Disjoint Unions and Higher-order list functions

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Recap
- Disjoint union (or Tagged Values)
  - Groups different kinds of values into a single set.

- Higher-order functions on lists
  - many list functions follow “standard” schemes
  - avoid (almost) identical code fragments by parameterizing functions with functions
  - higher-order list functions based on natural concepts
  - succinct declarations achievable using higher-order functions

Precedence and associativity rules for expressions

Operator	Association	Precedence
`**`	Associates to the right	highest
`* / %`	Associates to the left	
`+ -`	Associates to the left	
`= <> > >= < <=`	No association	
`&&`	Associates to the left	
`||`	Associates to the left	lowest

- a monadic operator has higher precedence than any dyadic
- higher (larger) precedence means earlier evaluation
- function application associates to the left
- abstraction (`fun x -> e`) extends as far to the right as possible

For example:

- `- 2 - 5 * 7 > 3 - 1` means `((-2)-(5*7)) > (3-1)`
- `fact 2 - 4` means `(fact 2) - 4`
- $e_1\ e_2\ e_3\ e_4$ means $((e_1\ e_2)\ e_3)\ e_4$
- `fun x -> x 2` means ????

Precedence and associativity rules for types

- infix type operators: `*` and `->`
- suffix type operator: `list`

Association rules:

- `*` has NO association
- `->` associates to the right

Precedence rules:

- The suffix operator `list` has highest precedence
- `*` has higher precedence than `->`

For example:

- `int*int*int list` means `(int*int*(int list))`
- `int->int->int->int` means `int->(int->(int->int))`
- `'a*'b->'a*'b list` means `('a*'b)->('a*('b list))`

Part I: Disjoint Sets – An Example

A *shape* is either a circle, a square, or a triangle

- the union of three disjoint sets

```
type Shape =
  | Circle of float                    (* 1 *)
  | Square of float                    (* 2 *)
  | Triangle of float*float*float;;    (* 3 *)
```

This declaration provides three rules for generating shapes:

- if r : `float`, then `Circle` r : `Shape` `(* 1 *)`
- if s : `float`, then `Square` s : `Shape` `(* 2 *)`
- if (a, b, c) : `float*float*float`,
  then `Triangle`(a, b, c) : `Shape` `(* 3 *)`

Such types are also called an algebraic data type:

```
Circle    :  float → Shape
Square    :  float → Shape
Triangle  :  float * float * float → Shape
```

Part I: Disjoint Sets – An Example (II)

The tags `Circle`, `Square` and `Triangle` are called

constructors

```
- Circle 2.0;;
> val it : Shape = Circle 2.0

- Triangle(1.0, 2.0, 3.0);;
> val it : Shape = Triangle(1.0, 2.0, 3.0)

- Square 4.0;;
> val it : Shape = Square 4.0
```

`Circle`, `Square` and `Triangle` are used to construct values of type `Shape`,

Constructors in Patterns

A shape-area function is declared

```
let area =
  function
  | Circle r          -> System.Math.PI * r * r
  | Square a          -> a * a
  | Triangle(a,b,c) ->
      let s = (a + b + c)/2.0
      sqrt(s*(s-a)*(s-b)*(s-c));;
> val area : Shape -> float
```

following the structure of shapes.

- a constructor only matches itself

$$
\begin{aligned}
& \texttt{area (Circle 1.2)} \\
\rightsquigarrow\ & (\texttt{System.Math.PI * r * r}, [\texttt{r} \mapsto \texttt{1.2}]) \\
\rightsquigarrow\ & \ldots
\end{aligned}
$$

Enumeration types – the months

Months are naturally defined using tagged values::

```
type Month = | January | February | March | April
             | May | June | July | August | September
             | October | November | December;;
```

The days-in-a-month function is declared by

```
let daysOfMonth = function
  | February                                -> 28
  | April | June | September | November -> 30
  | _                                       -> 31;;
val daysOfMonth : Month -> int
```

Observe: Constructors need not have arguments

The `option` type

```
type 'a option = None | Some of 'a
```

Distinguishes the cases "nothing" and "something".

predefined

The constructor `Some` and `None` are polymorphic:

```
Some false;;
val it : bool option = Some false

Some (1, "a");;
val it : (int * string) option = Some (1, "a")

None;;
val it : 'a option = None
```

Observe: type variables are allowed in declarations of algebraic types

Example: Find first position of element in a list

```
let rec findPosA p x =
    function
    | y::_ when x=y -> Some p
    | _::ys         -> findPosA (p+1) x ys
    | []            -> None;;
val findPosA : int -> 'a -> 'a list -> int option when ...
```

```
let findPos x ys = findPosA 0 x ys;;
val findPos : 'a -> 'a list -> int option when ...
```

Examples

```
findPos 4 [2 .. 6];;
val it : int option = Some 2

findPos 7 [2 .. 6];;
val it : int option = None

Option.get(findPos 4 [2 .. 6]);;
val it : int = 2
```

Exercise

A (teaching) room at DTU is either an auditorium or a databar:

- an auditorium is characterized by a location and a number of seats.
- a databar is characterized by a location, a number of computers and a number of seats.

Declare a type *Room*.

Declare a function:

$$seatCapacity : Room \to int$$

Declare a function

$$computerCapacity : Room \to int\ option$$

Part 2:Motivation

Higher-order functions are

- everywhere

  $\Sigma_{i=a}^{b} f(i),\ \frac{df}{dx},\ \{x \in A \mid P(x)\}, \ldots$

- powerful

  Parameterized modules, succinct code ...

HIGHER-ORDER FUNCTIONS ARE USEFUL

now down to earth

- Many recursive declarations follows the same schema.

For example:

```
let rec f = function
            | []    ->   ...
            | x::xs ->   ... f(xs) ...
```

Succinct declarations achievable using higher-order functions

Contents

- Higher-order list functions (in the library)
  - map
  - contains, exists, forall, filter, tryFind
  - foldBack, fold

Avoid (almost) identical code fragments by
parameterizing functions with functions

A simple declaration of a list function

A typical declaration following the structure of lists:

```
let rec posList = function
                  | []    -> []
                  | x::xs -> (x > 0)::posList xs;;
val posList : int list -> bool list

posList [4; -5; 6];;
val it : bool list = [true; false; true]
```

Applies the function `fun x -> x > 0` to each element in a list

Another declaration with the same structure

```
let rec addElems = function
                   | []         -> []
                   | (x,y)::zs -> (x+y)::addElems zs;;
val addElems : (int * int) list -> int list

addElems [(1,2) ;(3,4)];;
val it : int list = [3; 7]
```

Applies the addition function + to each pair of integers in a list

The function: `map`

Applies a function to each element in a list

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n] = [f(v_1); f(v_2); \ldots; f(v_n)]$$

Declaration

Library function

```
let rec map f = function
                | []    -> []
                | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

Succinct declarations can be achieved using map, e.g.

```
let posList = map (fun x -> x > 0);;
val posList : int list -> bool list

let addElems = map (fun (x,y) -> x+y);;
val addElems : (int * int) list -> int list
```

Exercise

Declare a function

$$\texttt{g}\ [x_1, \ldots, x_n] = [x_1^2 + 1, \ldots, x_n^2 + 1]$$

Remember

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n]\ =\ [f(v_1); f(v_2); \ldots; f(v_n)]$$

where

```
map: ('a -> 'b) -> 'a list -> 'b list
```

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

$$\texttt{exists}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \texttt{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec exists p = function
                   | []    -> false
                   | x::xs -> p x || exists p xs;;
val exists : ('a -> bool) -> 'a list -> bool
```

Example

```
exists (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = true
```

Exercise

Declare `contains` function using `exists`.

```
let contains x ys = exists ????? ;;
val contains : 'a -> 'a list -> bool when 'a : equality
```

Remember

$$\texttt{exists}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \texttt{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

where

```
exists: ('a -> bool) -> 'a list -> bool
```

contains is a Library function

Higher-order list functions: `forall`

Predicate: For every x in xs : $p(x)$.

$$\texttt{forall}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec forall p = function
                   | []    -> true
                   | x::xs -> p x && forall p xs;;
val forall : ('a -> bool) -> 'a list -> bool
```

Example

```
forall (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = false
```

Exercises

Declare a function

`disjoint` *xs ys*

which is true when there are no common elements in the lists *xs* and *ys*, and false otherwise.

Declare a function

`subset` *xs ys*

which is true when every element in the lists *xs* is in *ys*, and false otherwise.

Remember

$$\texttt{forall}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

where

```
forall : ('a -> bool) -> 'a list -> bool
```

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

`filter` *p xs* is the list of those elements *x* of *xs* where $p(x) =$ `true`.

Declaration

Library function

```
let rec filter p =
  function
  | []    -> []
  | x::xs -> if p x then x :: filter p xs
             else filter p xs;;
val filter : ('a -> bool) -> 'a list -> 'a list
```

Example

```
filter System.Char.IsLetter ['1'; 'p'; 'F'; '-'];;
val it : char list = ['p'; 'F']
```

where `System.Char.IsLetter` *c* is true iff
$c \in \{\text{'A'}, \ldots, \text{'Z'}\} \cup \{\text{'a'}, \ldots, \text{'z'}\}$

Exercise

Declare a function

`inter` *xs ys*

which contains the common elements of the lists *xs* and *ys* — i.e. their intersection.

Remember:
`filter` *p xs* is the list of those elements *x* of *xs* where $p(x) = $ `true`. where

```
filter: ('a -> bool) -> 'a list -> 'a list
```

Higher-order list functions: `tryFind`

$$\texttt{tryFind}\ p\ xs = \begin{cases} \texttt{Some}\ x & \text{for an element } x \text{ of } xs \text{ with } p(x) = \text{true} \\ \texttt{None} & \text{if no such element exists} \end{cases}$$

```
let rec tryFind p =
  function
  | x::xs when p x -> Some x
  | _::xs          -> tryFind p xs
  | _              -> None;;
val tryFind : ('a -> bool) -> 'a list -> 'a option
```

```
tryFind (fun x -> x>3) [1;5;-2;8];;
val it : int option = Some 5
```

Folding a function over a list (I)

Example: sum of absolute values:

```
let rec absSum = function
                 | []    -> 0
                 | x::xs -> abs x + absSum xs;;
val absSum : int list -> int

absSum [-2; 2; -1; 1; 0];;
val it : int = 6
```

Folding a function over a list (II)

```
let rec absSum = function
                 | []    -> 0
                 | x::xs -> abs x + absSum xs;;
```

Let $f\ x\ a$ abbreviate $\texttt{abs}\ x + a$ in the evaluation:

$$
\begin{array}{rl}
 & \texttt{absSum}\ [x_0; x_1; \ldots; x_{n-1}] \\
\rightsquigarrow & \texttt{abs}\ x_0 + (\texttt{absSum}\ [x_1; \ldots; x_{n-1}]) \\
= & f\ x_0\ (\texttt{absSum}\ [x_1; \ldots; x_{n-1}]) \\
\rightsquigarrow & f\ x_0\ (f\ x_1\ (\texttt{absSum}[x_2; \ldots; x_{n-1}])) \\
 & \vdots \\
\rightsquigarrow & f\ x_0\ (f\ x_1\ (\cdots (f\ x_{n-1}\ 0) \cdots))
\end{array}
$$

This repeated application of f is also called a folding of f.

Many functions follow such recursion and evaluation schemes

Higher-order list functions: `foldBack` (1)

Suppose that $\otimes$ is an infix function. Then

$$\texttt{foldBack}\ (\otimes)\ [a_0;\ a_1;\ \ldots\ ;\ a_{n-2};\ a_{n-1}]\ e_b = a_0 \otimes (a_1 \otimes (\ldots (a_{n-2} \otimes (a_{n-1} \otimes e_b))\ldots))$$

$$\texttt{List.foldBack}\ (+)\ [1;\ 2;\ 3]\ 0 = 1+(2+(3+0)) = 6$$
$$\texttt{List.foldBack}\ (-)\ [1;\ 2;\ 3]\ 0 = 1-(2-(3-0)) = 2$$

Using the cons operator gives the append function `@` on lists:

$$\texttt{foldBack (fun x rst -> x::rst)}\ [x_0;\ x_1;\ \ldots;\ x_{n-1}]\ \texttt{ys}$$
$$= x_0\texttt{::}(x_1\texttt{::}\ \ldots\ \texttt{::}(x_{n-1}\texttt{::ys})\ \ldots\ ))$$
$$= [x_0;\ x_1;\ \ldots;\ x_{n-1}]\ \texttt{@}\ \texttt{ys}$$

so we get:

```
let (@) xs ys = List.foldBack (fun x rst -> x::rst) xs ys;;
val ( @ ) : 'a list -> 'a list -> 'a list

[1;2] @ [3;4];;
val it : int list = [1; 2; 3; 4]
```

Declaration of `foldBack`

```
let rec foldBack f xlst e =
  match xlst with
  | x::xs -> f x (foldBack f xs e)
  | []    -> e;;
val foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
let absSum xs = foldBack (fun x a -> abs x + a)  xs 0;;

let length xs = foldBack (fun _ n -> n+1) xs 0;;

let map f xs = foldBack (fun x rs -> f x :: rs) xs [];;
```

Exercise: union of sets

Let an insertion function be declared by

```
let insert x ys = if List.contains x ys then ys
                  else x::ys;;
```

Declare a union function on sets, where a set is represented by a list without duplicated elements.

Remember:

$$\texttt{foldBack}\ (\oplus)\ [x_1; x_2; \ldots; x_n]\ b \rightsquigarrow x_1 \oplus (x_2 \oplus \cdots \oplus (x_n \oplus b) \cdots)$$

where

```
foldBack: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Higher-order list functions: `fold` (1)

Suppose that $\oplus$ is an infix function.

Then the `fold` function is defined by:

$$\texttt{fold}\ (\oplus)\ e_a\ [b_0;\ b_1;\ \ldots\ ;\ b_{n-2};\ b_{n-1}\ ] = ((\ldots((e_a \oplus b_0) \oplus b_1)\ldots) \oplus b_{n-2}) \oplus b_{n-1}$$

i.e. it applies $\oplus$ from left to right.

Examples:

$$\texttt{List.fold}\ (-)\ 0\ [1;\ 2;\ 3] = ((0-1)-2)-3 = -6$$
$$\texttt{List.foldBack}\ (-)\ [1;\ 2;\ 3]\ 0 = 1-(2-(3-0)) = 2$$

Higher-order list functions: `fold` (2)

```
let rec fold f e =
  function
  | x::xs -> fold f (f e x) xs
  | []    -> e;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Using cons in connection with fold gives the reverse function:

```
let rev xs  = fold (fun rs x -> x::rs) [] xs;;
```

This function has a linear execution time:

```
    rev [1;2;3]
⇝   fold   (fun ...)  []          [1;2;3]
⇝   fold   (fun ...)  (1::[])     [2;3]
⇝   fold   (fun ...)  [1]         [2;3]
⇝   fold   (fun ...)  (2::[1])    [3]
⇝   fold   (fun ...)  [2;1]       [3]
⇝   fold   (fun ...)  (3::[2;1])  []
⇝   fold   (fun ...)  [3;2;1]     []
⇝   [3;2;1]
```

Summary

Part I: Disjoint union (Algebraic data types)
– provides a mean to declare types containing different kinds of values

In Week 6 we extend the notion with recursive definition – provides a mean to declare types for finite trees
Part II: Higher-order list functions

- Many recursive declarations follows the same schema.

Succinct declarations achievable using higher-order functions

Contents

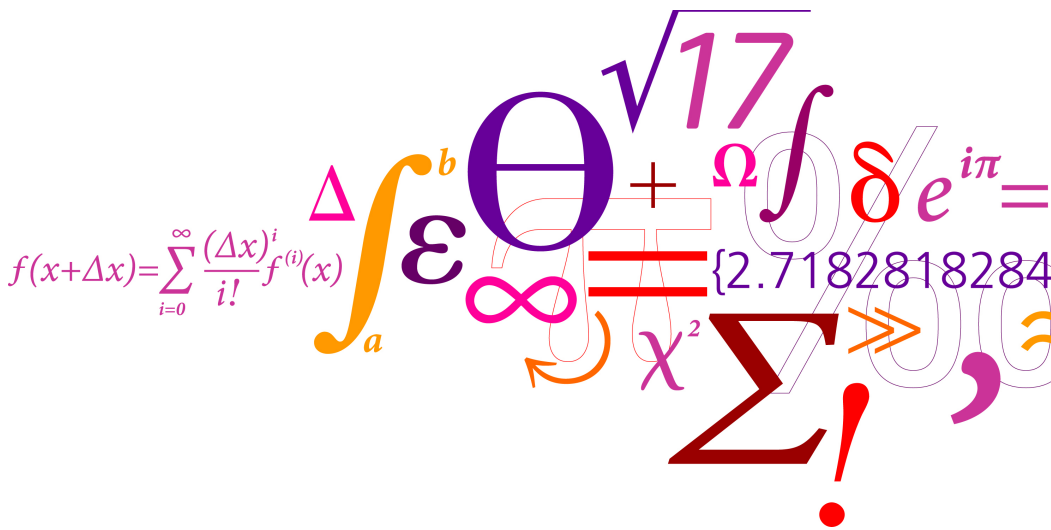
- Higher-order list functions (in the library)
  - map
  - contains, exists, forall, filter, tryFind
  - foldBack, fold

Avoid (almost) identical code fragments by parameterizing functions with functions

02157 Functional Programming

Collections: Finite Sets and Maps

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

Sets and Maps as abstract data types

- Useful when modelling
- Useful when programming
- Many similarities with the list library

Recommendation: Use these libraries whenever it is appropriate.

Succinct code through applications of higher-order functions

FSharp's immutable collections

- List: a finite sequence of elements of the same type
  - the sequence in which elements are enumerated is important
  - repetitions among elements of a list matters

- Set: a finite collection of elements of the same type
  - the sequence in which elements are enumerated is of no concern
  - repetitions among members of a set is of no concern

Today

- Map: a finite function from a domain of keys to values
  - the uniqueness of keys is an important property

Today

- Sequence: a possibly infinite sequence of elements of the same type
  - the elements of a sequence are computed by demand

Covered later in the semester

Types, data types and abstract data types

- A type is generated from basic types
  `int, float, bool, string, ...` and type variables
  `'a, 'b, 'c, ...` using type operators `*, ->, list, ...`

- A data type is characterized by
  - a type — `'alist`
  - a set of values — `[], [v], [v1; ...vn]`
  - a set of operations — `::, @ , List.rev, List.fold, ...`

- A abstract data type is a data type
  - where the representation of values is hidden — LiskovZilles 1974

Examples:

- `List` is a data type but not an abstract one
  — the representation of list values is visible ([] and ::)

- `Set` and `Map` are abstract data types

The set concept (1)

A *set* (in mathematics) is a collection of element like

$$\{\text{Bob}, \text{Bill}, \text{Ben}\}, \{1, 3, 5, 7, 9\}, \mathbb{N}, \text{and } \mathbb{R}$$

- the sequence in which elements are enumerated is of no concern, and
- repetitions among members of a set is of no concern either

It is possible to decide whether a given value is in the set.

$$\text{Alice} \notin \{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \text{and} \qquad 7 \in \{1, 3, 5, 7, 9\}$$

The empty set containing no element is written $\{\}$ or $\emptyset$.

The sets concept (2)

A set A is a *subset* of a set B, written $A \subseteq B$, if all the elements of A are also elements of B, for example

$$\{\text{Ben}, \text{Bob}\} \subseteq \{\text{Bob}, \text{Bill}, \text{Ben}\} \qquad \text{and} \qquad \{1, 3, 5, 7, 9\} \subseteq \mathbb{N}$$

Two sets A and B are equal, if they are both subsets of each other:

$$A = B \qquad \text{if and only if} \qquad A \subseteq B \text{ and } B \subseteq A$$

i.e. two sets are equal if they contain exactly the same elements.

The subset of a set A which consists of those elements satisfying a predicate p can be expressed using a *set-comprehension*:

$$\{x \in A \mid p(x)\}$$

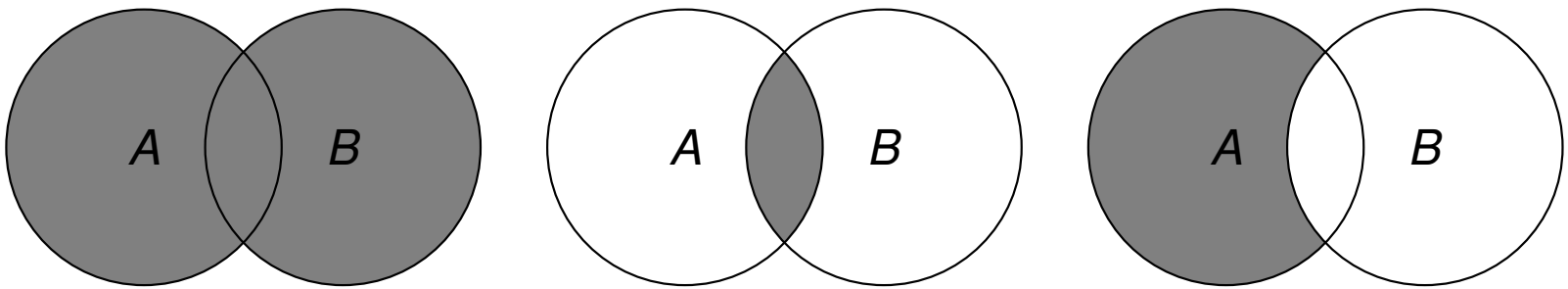
For example:

$$\{1, 3, 5, 7, 9\} = \{x \in \mathbb{N} \mid \text{odd}(x) \text{ and } x < 11\}$$

The set concept (3)

Some standard operations on sets:

$$
\begin{array}{lcll}
A \cup B & = & \{x \mid x \in A \text{ or } x \in B\} & \text{union} \\
A \cap B & = & \{x \mid x \in A \text{ and } x \in B\} & \text{intersection} \\
A \setminus B & = & \{x \in A \mid x \notin B\} & \text{difference}
\end{array}
$$



(a) $A \cup B$ (b) $A \cap B$ (c) $A \setminus B$

Figure: Venn diagrams for (a) union, (b) intersection and (c) difference

For example

$$
\begin{array}{lcl}
\{\text{Bob}, \text{Bill}, \text{Ben}\} \cup \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Alice}, \text{Ann}, \text{Bob}, \text{Bill}, \text{Ben}\} \\
\{\text{Bob}, \text{Bill}, \text{Ben}\} \cap \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Bill}\} \\
\{\text{Bob}, \text{Bill}, \text{Ben}\} \setminus \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Bob}, \text{Ben}\}
\end{array}
$$

Abstract Data Types

An Abstract Data Type: A type together with a collection of operations, where

- the representation of values is hidden.

An abstract data type for sets must have:

- Operations to generate sets from the elements. Why?
- Operations to extract the elements of a set. Why?
- Standard operations on sets.

An Abstract Data Type: `Set<a'>`

An abstract type for sets should at least support the following:

```
empty:      Set<'a>
add:        'a -> Set<'a> -> Set<a'>
union:      Set<'a> -> Set<'a> -> Set<a'>
intersect:  Set<'a> -> Set<'a> -> Set<a'>
difference: Set<'a> -> Set<'a> -> Set<a'>
contains:   'a -> Set<'a> -> bool
toList:     Set<'a> -> a' list
```

where

- any finite set can be generated by repeatedly adding elements to the empty set;
- union, intersection and difference are fundamental set operations;
- `contains` and `toList` are used to inspect the set

Note:

- the above operations are supported by the library `Set`.
- the representation of sets used by `Set` is hidden from the user.

Finite sets in F#

The `Set` library of F# supports finite sets. An efficient implementation is based on balanced binary trees.

Examples:

```
set ["Bob"; "Bill"; "Ben"];;
val it : Set<string> = set ["Ben"; "Bill"; "Bob"]

set [3; 1; 9; 5; 7; 9; 1];;
val it : Set<int> = set [1; 3; 5; 7; 9]
```

Equality of two sets is tested in the usual manner:

```
set["Bob";"Bill";"Ben"] = set["Bill";"Ben";"Bill";"Bob"];;
val it : bool = true
```

Sets are ordered on the basis of a lexicographical ordering:

```
compare (set ["Ann";"Jane"]) (set ["Bill";"Ben";"Bob"]);;
val it : int = -1
```

Immutability of `Set<'a>`

```
let s = Set.ofList [3; 2; 0];;
val s : Set<int> = set [0; 2; 3]

Set.add 1 s;;
val it : Set<int> = set [0; 1; 2; 3]

s;;
val it : Set<int> = set [0; 2; 3]
```

Evaluation of Set.add 1 s does not change the value of s.

Selected further operations (1)

- `ofList: 'a list -> Set<'a>`,
  where `ofList` $[a_0; \ldots; a_{n-1}] = \{a_0; \ldots; a_{n-1}\}$
- `remove: 'a -> Set<'a> -> Set<'a>`,
  where `remove` $a\ A = A \setminus \{a\}$
- `minElement: Set<'a> -> 'a`
  where `minElement` $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\} = a_0$ when $n > 0$
  (assuming that the enumeration respect the ordering)

Notice that minElement on a non-empty set is well-defined due to the ordering:

```
Set.minElement (Set.ofList ["Bob"; "Bill"; "Ben"]);;
val it : string = "Ben"
```

Selected further operations (2)

- `filter: ('a -> bool) -> Set<'a> -> Set<'a>`, where `filter` $p\ A = \{x \in A \mid p(x)\}$
- `exists: ('a -> bool) -> Set<'a> -> bool`, where `exists` $p\ A = \exists x \in A.p(x)$
- `forall: ('a -> bool) -> Set<'a> -> bool`, where `forall` $p\ A = \forall x \in A.p(x)$
- `fold: ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a`, where

$$\begin{aligned}&\texttt{fold}\ f\ a\ \{b_0, b_1, \ldots, b_{n-2}, b_{n-1}\}\\ &= f(f(f(\cdots f(f(a, b_0), b_1), \ldots), b_{n-2}), b_{n-1})\end{aligned}$$

These work similar to their List siblings, e.g.

`Set.fold (-) 0 (set [1; 2; 3])` $= ((0-1)-2)-3 = -6$

where the ordering is exploited.

Example: Map Coloring (1)

Maps and colors are modelled in a more natural way using sets:

```
type Country  = string;;
type Map      = Set<Country*Country>;;
type Color    = Set<Country>;;
type Coloring = Set<Color>;;
```

WHY?

The function:

```
areNb: Country -> Country -> Map -> bool
```

Two countries c_1, c_2 are neighbors in a map m,
if either $(c_1, c_2) \in m$ or $(c_2, c_1) \in m$:

```
let areNb c1 c2 m =  ?
```

Remember:

```
contains:   'a -> Set<'a> -> bool
exists: ('a -> bool) -> Set<'a> -> bool
```

Example: Map Coloring (2)

Maps and colors are modelled in a more natural way using sets:

```
type Country  = string;;
type Map      = Set<Country*Country>;;
type Color    = Set<Country>;;
type Coloring = Set<Color>;;
```

The function

```
canBeExtBy: Map -> Color -> Country -> bool
```

Color *col* and be extended by a country *c* given map *m*,
if for every country c' in *col*: *c* and c' are not neighbours in *m*

```
let canBeExtBy m col c =  ?
```

Remember

```
forall: ('a -> bool) -> Set<'a> -> bool
```

Example: Map Coloring (3)

The function

```
extColoring: Map -> Coloring -> Country -> Coloring
```

is declared as a recursive function over the coloring:

WHY not use a fold function?

```
let rec extColoring m cols c =
    if Set.isEmpty cols
    then Set.singleton (Set.singleton c)
    else let col = Set.minElement cols
         let cols' = Set.remove col cols
         if canBeExtBy m col c
         then Set.add (Set.add c col) cols'
         else Set.add col (extColoring m cols' c);;
```

Notice similarity to a list recursion:

- base case [] corresponds to the empty set
- for a recursive case x::xs, the head x corresponds to the minimal element col and the tail xs corresponds to the ”rests” set cols’

The list-based version is more efficient (why?) and better readable.

Example: Map Coloring (4)

Maps and colors are modelled in a more natural way using sets:

```
type Country  = string;;
type Map      = Set<Country*Country>;;
type Color    = Set<Country>;;
type Coloring = Set<Color>;;
```

A set of countries is obtained from a map by the function:

```
countries: Map -> Set<Country>
```

that is based on repeated insertion of the countries into a set:

```
let countries m =  ?
```

Remember

```
fold:     ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a
foldBack: ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b
```

Example: Map Coloring (5)

Maps and colors are modelled in a more natural way using sets:

```
type Country  = string;;
type Map      = Set<Country*Country>;;
type Color    = Set<Country>;;
type Coloring = Set<Color>;;
```

The function

```
colCntrs: Map -> Set<Country> -> Coloring
```

is based on repeated extension of colorings by countries using the `extColoring` function:

```
let colCntrs m cs =  ?
```

Remember

```
fold:     ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a
foldBack: ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b

extColoring: Map -> Coloring -> Country -> Coloring
```

Example: Map Coloring (6)

The function that creates a coloring from a map is declared using functional composition:

```
let colMap m = colCntrs m (countries m);;

let exMap = Set.ofList [("a","b"); ("c","d"); ("d","a")];;

colMap exMap;;
val it : Set<Set<string>>
       = set [set ["a"; "c"]; set ["b"; "d"]]
```

The map concept

A *map* from a set A to a set B is a *finite* subset A' of A together with a *function* m defined on A': $m : A' \to B$.
The set A' is called the *domain* of m: $\mathrm{dom}\, m = A'$.

A map m can be described in a tabular form:

a_0	b_0
a_1	b_1
$\vdots$	
a_{n-1}	b_{n-1}

- An element a_i in the set A' is called a *key*
- A pair (a_i, b_i) is called an *entry*, and
- b_i is called the *value* for the key a_i.

We denote the sets of entries of a map as follows:

$$\mathrm{entriesOf}(m) = \{(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\}$$

Selected map operations in F#

- `ofList: ('a*'b) list -> Map<'a,'b>`
  `ofList` $[(a_0, b_0); \ldots; (a_{n-1}, b_{n-1})] = m$

- `add: 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>`
  `add` $a\ b\ m = m'$, where m' is obtained m by overriding m with the entry (a, b)

- `find: 'a -> Map<'a,'b> -> 'b`
  `find` $a\ m = m(a)$, if $a \in \mathrm{dom}\ m$;
  otherwise an exception is raised

- `tryFind: 'a -> Map<'a,'b> -> 'b option`
  `tryFind` $a\ m =$ `Some` $(m(a))$, if $a \in \mathrm{dom}\ m$; `None` otherwise

- `foldBack:('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c`
  `foldBack` $f\ m\ c = f\ a_0\ b_0\ (f\ a_1\ b_1\ (f \ldots (f\ a_{n-1}\ b_{n-1}\ c) \cdots))$

Immutability of `Map<'Key,'Value>`

```
let m = Map.ofList [(0,"a") ; (2 "c"); (3,"d")];;
val m : Map<int,string> =
          map [(0, "a"); (2, "c"); (3, "d")]

Map.add 1 "b" m;;
val it : Map<int,string> =
          map [(0, "a"); (1, "b"); (2, "c"); (3, "d")]

Map.tryFind 1 m;;
val it : string option = None
```

Evaluation of Map.add 1 ”b” m does not change the value of m.

A few examples

```
let reg1 = Map.ofList [("a1",("cheese",25));
                       ("a2",("herring",4));
                       ("a3",("soft drink",5))];;
val reg1 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5))]
```

An entry can be added to a map using `add` and the value for a key in a map is retrieved using either `find` or `tryFind`:

```
let reg2 = Map.add "a4" ("bread", 6) reg1;;
val reg2 : Map<string,(string * int)> =
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));
       ("a3", ("soft drink", 5)); ("a4", ("bread", 6))]

Map.find "a2" reg1;;
val it : string * int = ("herring", 4)

Map.tryFind "a2" reg1;;
val it : (string * int) option = Some ("herring", 4)
```

An example using Map.foldBack

We can extract the list of article codes and prices for a given register using the fold functions for maps:

```
let reg1 = Map.ofList [("a1",("cheese",25));
                       ("a2",("herring",4));
                       ("a3",("soft drink",5))];;

Map.foldBack f reg1 [];;
val it : (string * int) list =
    [("a1", 25); ("a2", 4); ("a3", 5)]
```

What is f?

Remember

```
foldBack:('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c
```

The higher-order Map functions are similar to their List and Set siblings.

Example: Cash register (1)

```
type ArticleCode = string;;
type ArticleName = string;;
type NoPieces    = int;;
type Price       = int;;

type Info        = NoPieces * ArticleName * Price;;
type Infoseq     = Info list;;
type Bill        = Infoseq * Price;;
```

The natural model of a register is using a map:

```
type Register    = Map<ArticleCode, ArticleName*Price>;;
```

since an article code is *a unique identification* of an article.

First version:

```
type Item        = NoPieces * ArticleCode;;
type Purchase    = Item list;;
```

Example: Cash register (1) - a recursive program

```
exception FindArticle;;

(* makebill: Register -> Purchase -> Bill *)
let rec makeBill reg = function
    | []              -> ([],0)
    | (np,ac)::pur ->
        match Map.tryFind ac reg with
        | None                -> raise FindArticle
        | Some(aname,aprice) ->
            let tprice          = np*aprice
            let (infos,sumbill) = makeBill reg pur
            ((np,aname,tprice)::infos, tprice+sumbill);;

let pur = [(3,"a2"); (1,"a1")];;
makeBill reg1 pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

- the lookup in the register is managed by a `Map.tryFind`

Example: Cash register (2) - using List.foldBack

```
let makeBill' reg pur =
   let f (np,ac) (infos,billprice)
          = let (aname, aprice) = Map.find ac reg
            let tprice          = np*aprice
            ((np,aname,tprice)::infos, tprice+billprice)
   List.foldBack f pur ([],0);;

makeBill' reg1 pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

- the recursion is handled by List.foldBack
- the exception is handled by `Map.find`

Example: Cash register (2) - using maps for purchases

The purchase: 3 herrings, one piece of cheese, and 2 herrings, is the same as a purchase of one piece of cheese and 5 herrings.

A purchase associated number of pieces with article codes:

```
type Purchase    = Map<ArticleCode,NoPieces>;;
```

A bill is produced by folding a function over a map-purchase:

```
let makeBill'' reg pur =
   let f ac np (infos,billprice)
         = let (aname, aprice) = Map.find ac reg
           let tprice          = np*aprice
           ((np,aname,tprice)::infos, tprice+billprice)
   Map.foldBack f pur ([],0);;

let purMap = Map.ofList [("a2",3); ("a1",1)];;
val purMap : Map<string,int> = map [("a1", 1); ("a2", 3)]

makeBill'' reg1 purMap;;
val it = ([(1, "cheese", 25); (3, "herring", 12)], 37)
```

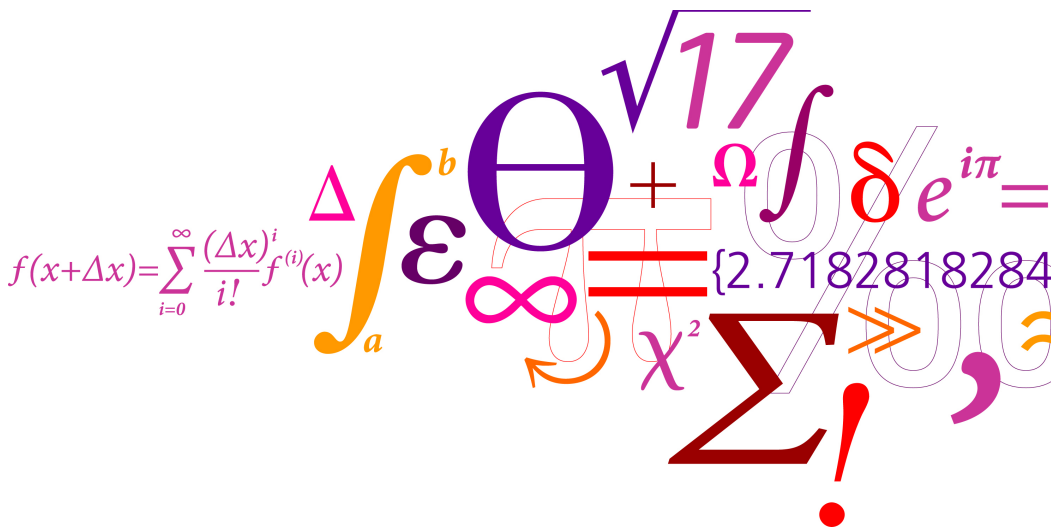
Summary

- The concepts of sets and maps.
- Fundamental operations on sets and maps.
- Applications of sets and maps.

02157 Functional Programming

Finite Trees (I)

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

Finite Trees

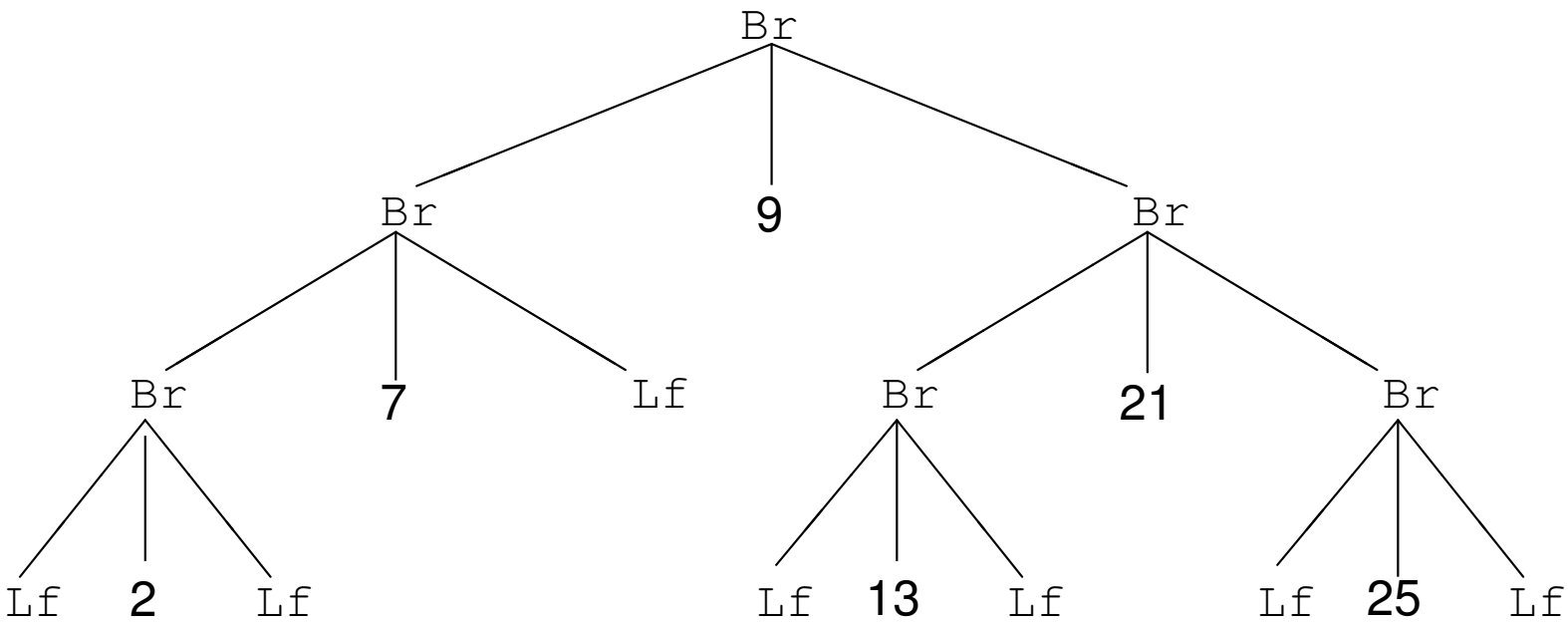
- recursive declarations of algebraic types
- meaning of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a fixed branching structure
- trees with a variable number of sub-trees
- illustrative examples

Mutually recursive type and function declarations

Finite trees

A *finite tree* is a value containing subcomponents of the same type

Example: A *binary tree*



A tree is a connected, acyclic, undirected graph, where

- the top node (carrying value 9) is called the root
- a branch node has two children — constructor Br
- a node without children is called a leaf — constructor Lf

Example: Binary Trees

A *recursive datatype* is used to represent values that are trees.

```
type Tree = | Lf
            | Br of Tree*int*Tree;;
```

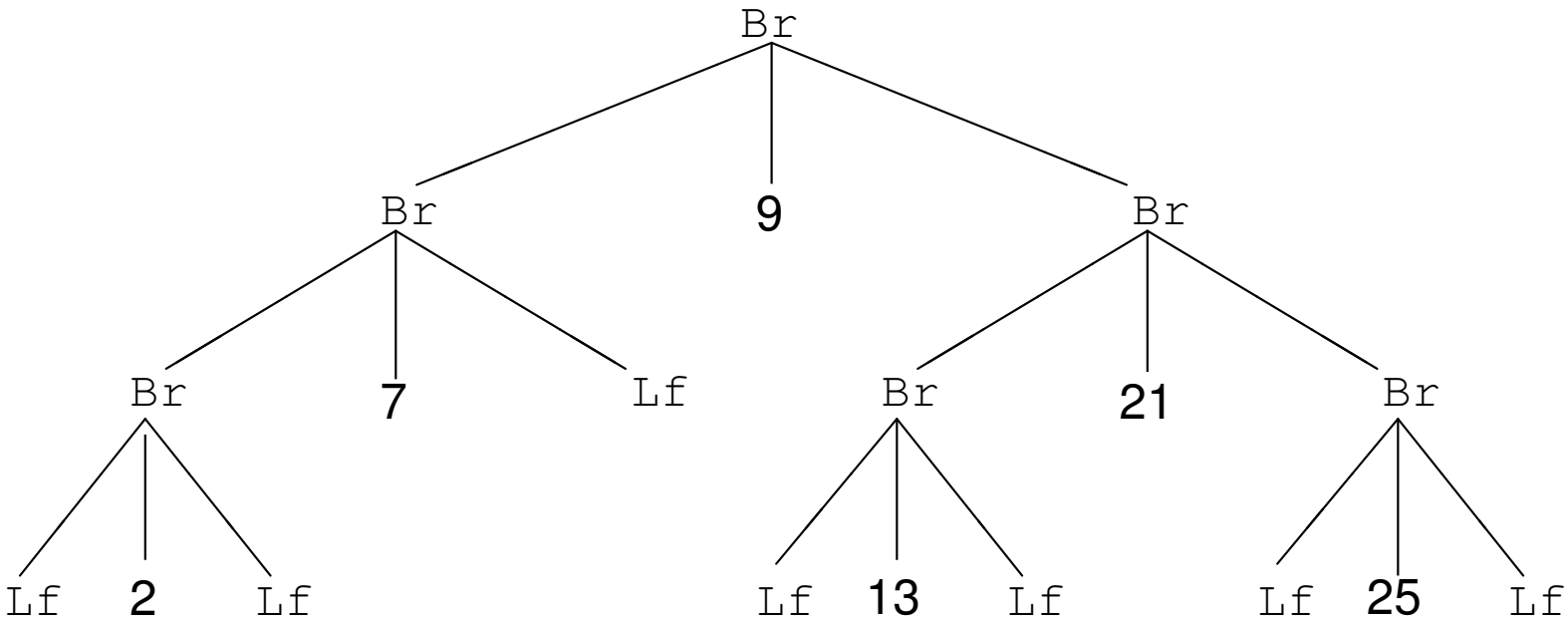
The declaration provides rules for generating trees:

1. `Lf` is a tree
2. if t_1, t_2 are trees and n is an integer, then `Br`(t_1, n, t_2) is a tree.
3. the type Tree contains no other values than those generated by repeated use of Rules 1. and 2.

The tags Lf and Br are called constructors:

```
Lf  :  Tree
Br  :  Tree * int * Tree → Tree
```

Example: Binary Trees



Corresponding F#-value:

```
Br(Br(Br(Lf,2,Lf),7,Lf),
   9,
   Br(Br(Lf,13,Lf),21,Br(Lf,25,Lf)))
```

Traversals of binary trees

- Pre-order traversal: First visit the root node, then traverse the left sub-tree in pre-order and finally traverse the right sub-tree in pre-order.
- In-order traversal: First traverse the left sub-tree in in-order, then visit the root node and finally traverse the right sub-tree in in-order.
- Post-order traversal: First traverse the left sub-tree in post-order, then traverse the right sub-tree in post-order and finally visit the root node.

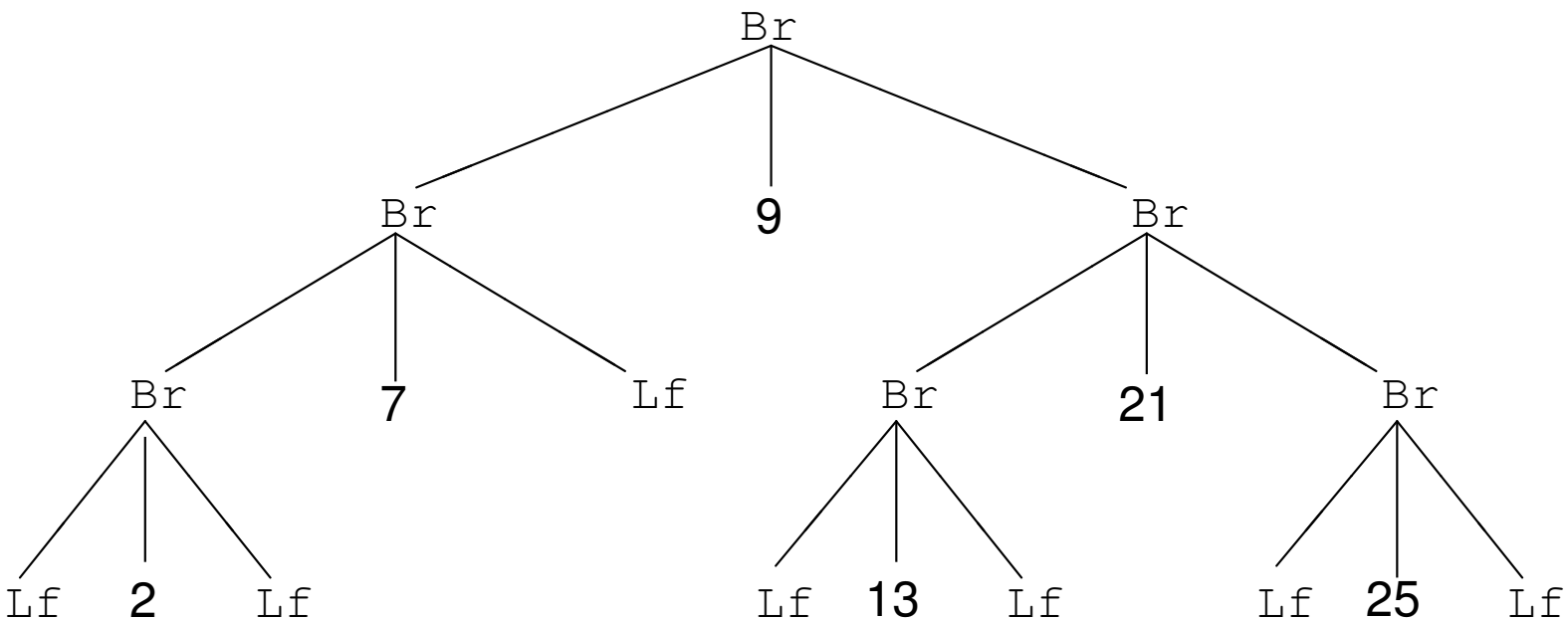
In-order traversal

```
let rec inOrder =
    function
    | Lf           -> []
    | Br(t1,j,t2) -> inOrder t1 @ [j] @ inOrder t2;;
val toList : Tree -> int list

inOrder(Br(Br(Lf,1,Lf), 3, Br(Br(Lf,4,Lf), 5, Lf)));;
val it : int list = [1; 3; 4; 5]
```

Binary search tree

Condition: for every node containing the value x: every value in the left subtree is smaller then x, and every value in the right subtree is greater than x.

Example: A *binary search tree*



Binary search trees: Insertion

- Recursion following the structure of trees
- Constructors `Lf` and `Br` are used in patterns to decompose a tree into its parts
- Constructors `Lf` and `Br` are used in expressions to construct a tree from its parts
- The search tree condition is an invariant for `insert`

```
let rec insert i =
    function
    | Lf                  ->  Br(Lf,i,Lf)
    | Br(t1,j,t2) as tr -> // Layered pattern
          match compare i j with
          | 0             -> tr
          | n when n<0    -> Br(insert i t1 , j, t2)
          | _             -> Br(t1,j, insert i t2);;
val insert : int -> Tree -> Tree
```

Example:

```
let t1 = Br(Lf, 3, Br(Lf, 5, Lf));;
let t2 = insert 4 t1;;
val t2 : Tree = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))
```

Binary search trees: `contains`

```
let rec contains  i =
    function
    | Lf                    -> false
    | Br(_,j,_)   when i=j -> true
    | Br(t1,j,_) when i<j -> contains i t1
    | Br(_,j,t2)           -> contains i t2;;
val contains : int -> Tree -> bool

let t = Br(Br(Br(Lf,2,Lf),7,Lf),
           9,
           Br(Br(Lf,13,Lf),21,Br(Lf,25,Lf)));;

contains 21 t;;
val it : bool = true

contains 4 t;;
val it : bool = false
```

Parameterize type declarations

The programs on search trees require only an ordering on elements – they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = | Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program text is unchanged (though polymorphic now), for example

```
let rec insert i = function
        ....
  | Br(t1,j,t2) as tr -> match compare i j with
        .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4  (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
       = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```

So far

- Declaration of a recursive algebraic data type, that is, a type for a finite tree
- Meaning of the type declaration in the form of rules for generating values
- typical functions on trees:
  - gathering information from a tree — Example: inOrder
  - inspecting a tree — Example: contains
  - constructs of a new tree — Example: insert

Manipulation of arithmetical expressions

Consider $f(x)$:

$$3 \cdot (x - 1) - 2 \cdot x$$

We may be interested in

- computation of values, e.g. $f(2)$
- differentiation, e.g. $f'(x) = (3 \cdot 1 + 0 \cdot (x - 1)) - (2 \cdot 1 + 0 \cdot x)$
- simplification of the expressions, e.g. $f'(x) = 1$
- .....

We would like a suitable representation of such arithmetical expressions that supports the above manipulations

How would you visualize the expressions as a tree?

- root?
- leaves?
- branches?

Example: Expression Trees

```
type Fexpr =
  | Const of float
  | X
  | Add of Fexpr * Fexpr
  | Sub of Fexpr * Fexpr
  | Mul of Fexpr * Fexpr
  | Div of Fexpr * Fexpr;;
```

Defines 6 constructors:

- `Const: float -> Fexpr`
- `X : Fexpr`
- `Add: Fexpr * Fexpr -> Fexpr`
- `Sub: Fexpr * Fexpr -> Fexpr`
- `Mul: Fexpr * Fexpr -> Fexpr`
- `Div: Fexpr * Fexpr -> Fexpr`

- Can you write 3 values of type Fexpr?
- Drawings of trees?

Expressions: Computation of values

Given a value (a float) for `X`, then every expression denote a float.

```
compute : float -> Fexpr -> float
```

```
let rec compute x =
   function
    | Const r        -> r
    | X              -> x
    | Add(fe1,fe2)   -> compute x fe1 + compute x fe2
    | Sub(fe1,fe2)   -> compute x fe1 - compute x fe2
    | Mul(fe1,fe2)   -> compute x fe1 * compute x fe2
    | Div(fe1,fe2)   -> compute x fe1 / compute x fe2;;
```

Example:

```
compute 4.0 (Mul(X, Add(Const 2.0, X)));;
val it : float = 24.0
```

Blackboard exercise: Substitution

```
type Fexpr = | Const of float
             | X
             | Add of Fexpr * Fexpr
             | Sub of Fexpr * Fexpr
             | Mul of Fexpr * Fexpr
             | Div of Fexpr * Fexpr;;
```

Declare a function

```
substX: Fexpr -> Fexpr -> Fexpr
```

so that `substX` e' e is the expression obtained from e by substituting every occurrence of `X` with e'

For example:

```
let ex = Add(Sub(X, Const 2.0), Mul(Const 4.0, X));;

substX (Div(X,X)) ex;;
val it : Fexpr =
   Add(Sub(Div(X,X), Const 2.0), Mul(Const 4.0, Div(X,X)))
```

Symbolic Differentiation `D: Fexpr -> Fexpr`

A classic example in functional programming:

```
let rec D = function
  | Const _       -> Const 0.0
  | X             -> Const 1.0
  | Add(fe1,fe2) -> Add(D fe1,D fe2)
  | Sub(fe1,fe2) -> Sub(D fe1,D fe2)
  | Mul(fe1,fe2) -> Add(Mul(D fe1,fe2),Mul(fe1,D fe2))
  | Div(fe1,fe2) -> Div(
                        Sub(Mul(D fe1,fe2),Mul(fe1,D fe2)),
                        Mul(fe2,fe2));;
```

Notice the direct correspondence with the rules of differentiation.

Can be tried out directly, as tree are ”just” values, for example:

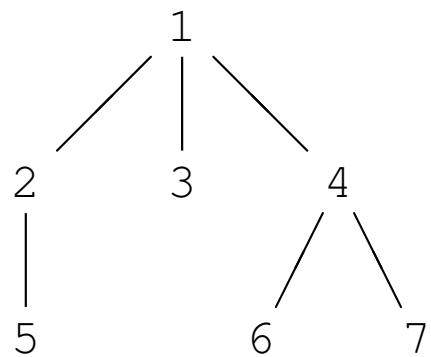
```
D(Add(Mul(Const 3.0, X), Mul(X, X)));;
val it : Fexpr =
  Add
    (Add (Mul (Const 0.0,X),Mul (Const 3.0,Const 1.0)),
     Add (Mul (Const 1.0,X),Mul (X,Const 1.0)))
```

Trees with a variable number of sub-trees

An archetypical declaration:

```
type ListTree<'a> = Node of 'a * (ListTree<'a> list)
```

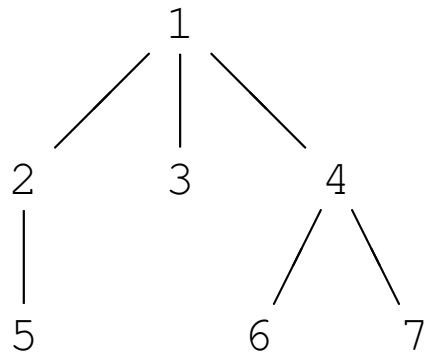
- `Node(`x`,[])` represents a leaf tree containing the value x
- `Node(`x`,[`t_0`;...;`t_{n-1}`])` represents a tree with value x in the root and with n sub-trees represented by the values $t_0, \ldots, t_{n-1}$



It is represented by the value `t1` where

```
let t7 = Node(7,[]);;     let t6 = Node(6,[]);;
let t5 = Node(5,[]);;     let t3 = Node(3,[]);;
let t2 = Node(2,[t5]);;   let t4 = Node(4,[t6; t7]);;
let t1 = Node(1,[t2; t3; t4]);;
```

Depth-first traversal of a ListTree

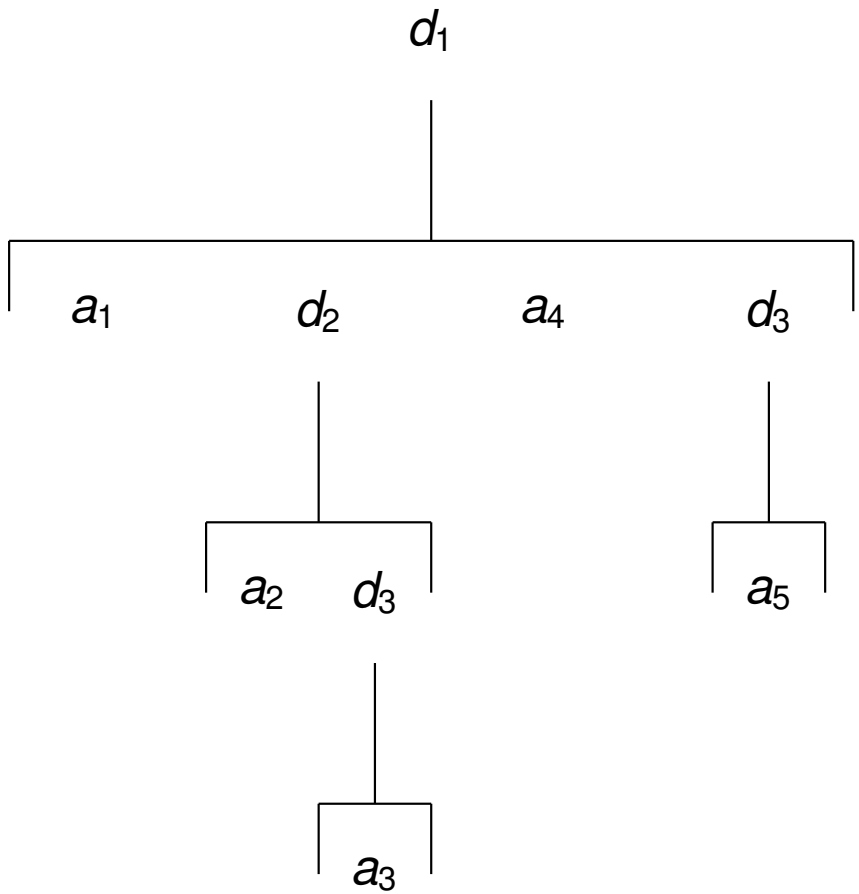


Corresponds to the following order of the elements: 1, 2, 5, 3, 4, 6, 7

Invent a more general function traversing a list of List trees:

```
let rec depthFirstList =
   function
   | [] -> []
   | Node(n,ts)::trest -> n::depthFirstList(ts @ trest)
depthFirstList : ListTree<'a> list -> 'a list

let depthFirst t = depthFirstList [t]
 depthFirst1 : t:ListTree<'a> -> 'a list

depthFirst t1;;
val it : int list = [1; 2; 5; 3; 4; 6; 7]
```

Mutual recursion. Example: File system



- A file system is a list of elements
- an element is a file or a directory, which is a named file system

We focus on structure now – not on file content

Mutually recursive type declarations

- are combined using and

```
type FileSys = Element list
and Element =
  | File of string
  | Dir of string * FileSys


let d1 =
  Dir("d1",[File "a1";
            Dir("d2", [File "a2";
                       Dir("d3", [File "a3"])]);
            File "a4";
            Dir("d3", [File "a5"])
           ])
```

The type of d1 is ?

Mutually recursive function declarations

- are combined using and

Example: extract the names occurring in file systems and elements.

```
let rec namesFileSys =
  function
  | []    -> []
  | e::es -> (namesElement e) @ (namesFileSys es)
and namesElement =
  function
  | File s    -> [s]
  | Dir(s,fs) -> s :: (namesFileSys fs) ;;
val namesFileSys : Element list -> string list
val namesElement : Element -> string list


namesElement d1 ;;
val it : string list = ["d1"; "a1"; "d2"; "a2";
                        "d3"; "a3"; "a4"; "d3"; "a5"]
```

Summary

Finite Trees

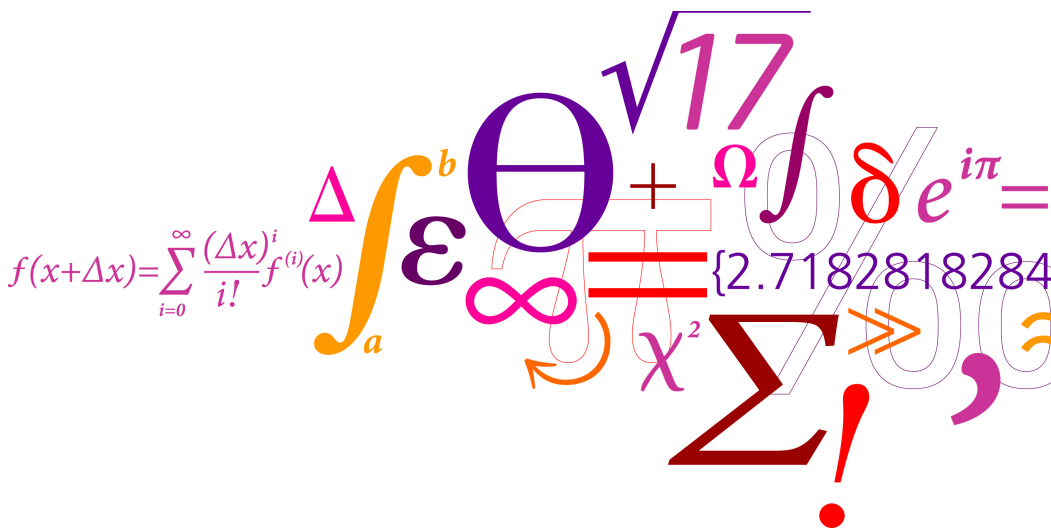
- recursive declarations of algebraic types
- meaning of type declarations: rules generating values
- typical recursions following the structure of trees
- trees with a fixed branching structure
- trees with a variable number of sub-trees
- illustrative examples

Mutually recursive type and function declarations

02157 Functional Programming

Interpreters for two simple languages
– including exercises

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Miscellaneous
  - On type declarations — Type abbreviations and ”real” types
- Finite trees: Two examples
  - An interpreter for a simple expression language
  - An interpreter for a simple while-language

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

This type is different from the type `T2`:

```
type T2 = | A of int | B of bool;;

let v1 = T1.A(2);;
let v2 = T2.A(2);;
```

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

This type is different from the type `T2`:

```
type T2 = | A of int | B of bool;;

let v1 = T1.A(2);;
let v2 = T2.A(2);;

v1=v2;;
... error ... : This expression was expected to have type
    T1
but here has type
    T2
```

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

This type is different from the type `T2`:

```
type T2 = | A of int | B of bool;;

let v1 = T1.A(2);;
let v2 = T2.A(2);;

v1=v2;;
... error ... : This expression was expected to have type
    T1
but here has type
    T2
```

- Values of type `T1` and `T2` cannot be compared — the types are, in fact, different.
- A similar observation applies for records.

On type declarations (2)

The other kinds of type declarations we have considered so far define type abbreviations. For example, the types `TA1` and `TA2`:

```
type TA1 = int * bool;;
type TA2 = int * bool;;
```

are identical and just shorthands for `int * bool`.

On type declarations (2)

The other kinds of type declarations we have considered so far define type abbreviations. For example, the types `TA1` and `TA2`:

```
type TA1 = int * bool;;
type TA2 = int * bool;;
```

are identical and just shorthands for `int * bool`.

For example:

```
let v1 = (1,true):TA1;;
val v1 : TA1 = (1, true)
let v2 = (1,true):TA2;;
val v2 : TA2 = (1, true)
let v3 = (1,true):int*bool;;
val v3 : int * bool = (1, true)

v1=v2;;
val it : bool = true

v2=v3;;
val it : bool = true
```

On type declarations (3)

Type declarations, including type abbreviations, often have a pragmatic role. For example, succinct way of presenting a model:

```
type CourseNo    = int
type Title       = string
type ECTS        = int
type CourseDesc = Title * ECTS

type CourseBase = Map<CourseNo, CourseDesc>

type Mandatory   = Set<CourseNo>
type Optional    = Set<CourseNo>
type CourseGroup = Mandatory * Optional
```

On type declarations (3)

Type declarations, including type abbreviations, often have a pragmatic role. For example, succinct way of presenting a model:

```
type CourseNo   = int
type Title      = string
type ECTS       = int
type CourseDesc = Title * ECTS

type CourseBase = Map<CourseNo, CourseDesc>

type Mandatory   = Set<CourseNo>
type Optional    = Set<CourseNo>
type CourseGroup = Mandatory * Optional
```

or the purpose of a function like

```
isValidCourseGroup: CourseGroup -> CourseBase -> bool
```

On type declarations (3)

Type declarations, including type abbreviations, often have a pragmatic role. For example, succinct way of presenting a model:

```
type CourseNo    = int
type Title       = string
type ECTS        = int
type CourseDesc = Title * ECTS

type CourseBase = Map<CourseNo, CourseDesc>

type Mandatory   = Set<CourseNo>
type Optional    = Set<CourseNo>
type CourseGroup = Mandatory * Optional
```

or the purpose of a function like

```
isValidCourseGroup: CourseGroup -> CourseBase -> bool
```

The expanded type for this function is NOT “enriching”:

```
(Set<int>*Set<int>) -> Map<int,string*int> -> bool
```

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax

succinct programs, fast prototyping

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax

succinct programs, fast prototyping

The interpreter for the simple expression language is a higher-order function:

$$eval : Program \to Environment \to Value$$

The interpreter for a simple imperative programming language is a higher-order function:

$$I : Program \to State \to State$$

Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

The abstract syntax is defined by an algebraic datatype:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum   of ExprTree * ExprTree
                | Diff  of ExprTree * ExprTree
                | Prod  of ExprTree * ExprTree
                | Let of string * ExprTree * ExprTree;;
```

Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

The abstract syntax is defined by an algebraic datatype:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum   of ExprTree * ExprTree
                | Diff  of ExprTree * ExprTree
                | Prod  of ExprTree * ExprTree
                | Let of string * ExprTree * ExprTree;;
```

Example:

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

A tree `Let(`s`,`t_1`,`t_2`)` is evaluated in an environment *env*:

1. Evaluate t_1 to value v_1 in environment *env*.
2. Evaluate t_2 in *env* extended with the binding of s to v_1.

Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

A tree `Let(`s`,`t_1`,`t_2`)` is evaluated in an environment *env*:

1. Evaluate t_1 to value v_1 in environment *env*.
2. Evaluate t_2 in *env* extended with the binding of s to v_1.

An evaluation function

```
eval: ExprTree -> Map<string,int> -> int
```

is defined as follows:

```
let rec eval t env =
    match t with
    | Const n       -> n
    | Ident s       -> Map.find s env
    | Minus t       -> - (eval t env)
    | Sum(t1,t2)    -> eval t1 env + eval t2 env
    | Diff(t1,t2)   -> eval t1 env - eval t2 env
    | Prod(t1,t2)   -> eval t1 env * eval t2 env
    | Let(s,t1,t2) -> let v1   = eval t1 env
                      let env1 = Map.add s v1 env
                      eval t2 env1;;
```

Example

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

Example

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

Example

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

```
let env = Map.add "a" -7 Map.empty;;
eval et env;;
val it : int = 35
```

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Typical ingredients

- Arithmetical expressions

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Typical ingredients

- Arithmetical expressions
- Boolean expressions

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Typical ingredients

- Arithmetical expressions
- Boolean expressions
- Statements (assignments, sequential composition, loops, . . .

Arithmetic Expressions

- The declaration of the abstract syntax for Arithmetical Expressions

```
type AExp =       (* Arithmetical expressions *)
    | N  of int               (* numbers        *)
    | V  of string            (* variables      *)
    | Add of AExp * AExp      (* addition       *)
    | Mul of AExp * AExp      (* multiplication *)
    | Sub of AExp * AExp;;    (* subtraction    *)
```

Arithmetic Expressions

- The declaration of the abstract syntax for Arithmetical Expressions

```
type AExp =          (* Arithmetical expressions *)
   | N  of int               (* numbers          *)
   | V  of string            (* variables        *)
   | Add of AExp * AExp      (* addition         *)
   | Mul of AExp * AExp      (* multiplication   *)
   | Sub of AExp * AExp;;    (* subtraction      *)
```

  You do not need parenthesis, precedence rules, ect for the abstract syntax — you work directly on trees.

Semantics of Arithmetic Expressions

- A state maps variables to integers

```
type State = Map<string,int>;;
```

Semantics of Arithmetic Expressions

- A state maps variables to integers

```
type State = Map<string,int>;;
```

- The meaning of an expression is a function:

```
A: AExp -> State -> int
```

Semantics of Arithmetic Expressions

- A state maps variables to integers

```
type State = Map<string,int>;;
```

- The meaning of an expression is a function:

```
A: AExp -> State -> int
```

  defined inductively on the structure of arithmetic expressions

```
let rec A a s       =
    match a with
      | N n           -> n
      | V x           -> Map.find x s
      | Add(a1, a2) -> A a1 s + A a2 s
      | Mul(a1, a2) -> A a1 s * A a2 s
      | Sub(a1, a2) -> A a1 s - A a2 s;;
```

Boolean Expressions

- Abstract syntax

```
type BExp =                (* Boolean expressions *)
    | TT                         (* true          *)
    | FF                         (* false         *)
    | Eq of ....                 (* equality      *)
    | Lt of ....                 (* less than     *)
    | Neg of ....                (* negation      *)
    | Con of ....          ;;    (* conjunction   *)
```

Boolean Expressions

- Abstract syntax

```
type BExp =                (* Boolean expressions *)
    | TT                         (* true          *)
    | FF                         (* false         *)
    | Eq of ....                 (* equality      *)
    | Lt of ....                 (* less than     *)
    | Neg of ....                (* negation      *)
    | Con of ....          ;;    (* conjunction   *)
```

- Semantics B : BExp → State → bool

```
let rec B b s =
   match b with
    | TT       -> true
    | ....
```

Statements: Abstract Syntax

```
type Stm  =                          (* statements                *)
    | Ass of string * AExp           (* assignment                *)
    | Skip
    | Seq  of Stm * Stm              (* sequential composition    *)
    | ITE   of BExp * Stm * Stm      (* if-then-else              *)
    | While of BExp * Stm;;          (* while                     *)
```

Statements: Abstract Syntax

```
type Stm  =                          (* statements                *)
    | Ass of string * AExp           (* assignment                *)
    | Skip
    | Seq  of Stm * Stm              (* sequential composition    *)
    | ITE   of BExp * Stm * Stm      (* if-then-else              *)
    | While of BExp * Stm;;          (* while                     *)
```

Example of concrete syntax:

```
y:=1 ; while not(x=0) do (y:= y*x ; x:=x-1)
```

Abstract syntax ?

Update of states

An imperative program performs a sequence of state updates.

- The expression

  $$\texttt{update}\ y\ v\ s$$

  is the state that is as s except that y is mapped to v.

Update of states

An imperative program performs a sequence of state updates.

- The expression

  $$\texttt{update}\ y\ v\ s$$

  is the state that is as s except that y is mapped to v.
  Mathematically:

$$(update\ y\ v\ s)(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}$$

Update of states

An imperative program performs a sequence of state updates.

- The expression

  $$\texttt{update}\ y\ v\ s$$

  is the state that is as s except that y is mapped to v.
  Mathematically:

$$(update\ y\ v\ s)(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}$$

- Update is a higher-order function with the declaration:

```
let update x v s = Map.add x v s;;
```

- Type?

Interpreter for Statements

- The meaning of statements is a function

$$I : Stm \to State \to State$$

that is defined by induction on the structure of statements:

```
let rec I stm s =
    match stm with
    | Ass(x,a)          -> update x ( ... ) s
    | Skip              -> ...
    | Seq(stm1, stm2)   -> ...
    | ITE(b,stm1,stm2)  -> ...
    | While(b, stm)     -> ... ;;
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)


let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)


let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;


(* Define an initial state                                    *)
let s0 = Map.ofList [("x",4)];;
val s0 : Map<string,int> = map [("x", 4)]
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)


let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;


(* Define an initial state                                    *)
let s0 = Map.ofList [("x",4)];;
val s0 : Map<string,int> = map [("x", 4)]


(* Interpret the program                                      *)
let s1 = I fac s0;;
val s1 : Map<string,int> = map [("x", 1); ("y", 24)]
```

Exercises

- Complete the program skeleton for the interpreter, and try some examples.
- Extend the abstract syntax and the interpreter with if-then and repeat-until statements.
- Suppose that an expression of the form $inc(x)$ is added. It adds one to the value of x in the current state, and the value of the expression is this new value of x.

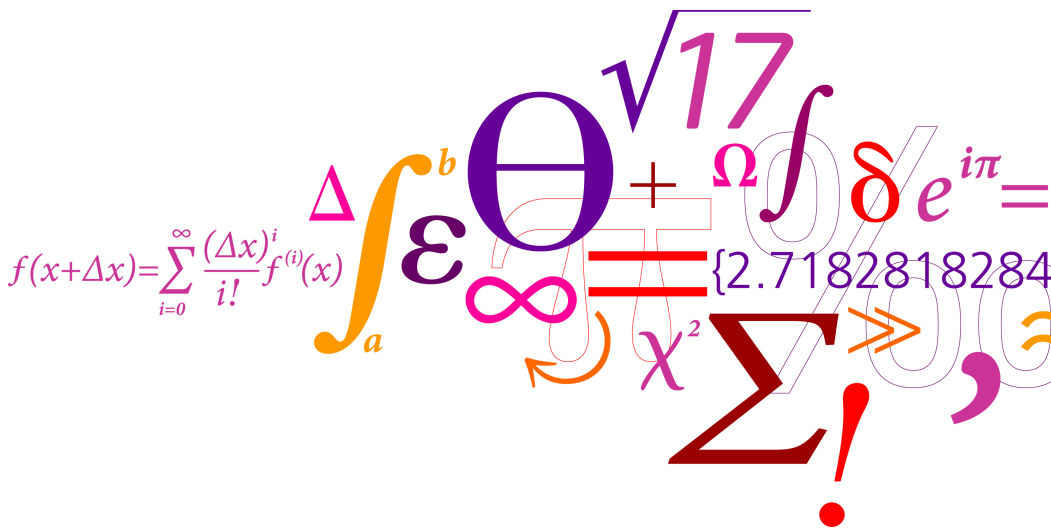
  How would you refine the interpreter to cope with this construct?

  - Analyse the problem and state the types for the refined interpretation functions

02157 Functional Programming

Lecture 8: Tail-recursive (iterative) functions (I)

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, for example, in order
  - to avoid evaluations with a huge amount of pending operations, e.g.

$$7+(6+(5\cdots+f\ 2\cdots))$$

  - to avoid inadequate use of `@` in recursive declarations.
- Iterative functions with accumulating parameters correspond to while-loops

An example: Factorial function (I)

Consider the following declaration:

```
let rec fact =
    function
    | 0 -> 1
    | n -> n * fact(n-1);;
val fact : int -> int
```

- What resources are needed to compute `fact`(N)?

Considerations:

- Computation time: number of individual computation steps.
- Space: the maximal memory needed during the computation to represent expressions and bindings.

An example: Factorial function (II)

Evaluation:

$$
\begin{array}{ll}
 & \texttt{fact}(N) \\
\rightsquigarrow & (\texttt{n * fact(n-1)}\ ,\ [\texttt{n} \mapsto N]) \\
\rightsquigarrow & N * \texttt{fact}(N-1) \\
\rightsquigarrow & N * (\texttt{n * fact(n-1)}\ ,\ [\texttt{n} \mapsto N-1]) \\
\rightsquigarrow & N * ((N-1) * \texttt{fact}(N-2)) \\
\vdots & \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * (2 * 1))) \cdots))) \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * 2)) \cdots))) \\
\vdots & \\
\rightsquigarrow & N!
\end{array}
$$

Time and space demands: proportional to N Is this satisfactory?

Another example: Naive reversal (I)

```
let rec naiveRev =
    function
    | []    -> []
    | x::xs -> naiveRev xs @ [x];;
val naiveRev : 'a list -> 'a list
```

Evaluation of `naiveRev` $[x_1, x_2, \ldots, x_n]$:

$$
\begin{aligned}
& \texttt{naiveRev}\,[x_1, x_2, \ldots, x_n] \\
\rightsquigarrow\ & \texttt{naiveRev}\,[x_2, \ldots, x_n]\,\texttt{@}\,[x_1] \\
\rightsquigarrow\ & (\texttt{naiveRev}\,[x_3, \ldots, x_n]\,\texttt{@}\,[x_2])\,\texttt{@}\,[x_1] \\
& \vdots \\
\rightsquigarrow\ & ((\cdots((\texttt{[]}\,\texttt{@}\,[x_n])\,\texttt{@}\,[x_{n-1}])\,\texttt{@}\cdots\texttt{@}\,[x_2])\,\texttt{@}\,[x_1])
\end{aligned}
$$

Space demands: proportional to n — satisfactory

Time demands: proportional to n^2 — not satisfactory

Examples: Accumulating parameters

Efficient solutions are obtained by using *more general functions*:

$$\begin{array}{lcl} \texttt{factA}(n, m) & = & n! \cdot m, \text{ for } n \geq 0 \\ \texttt{revA}(\texttt{[}x_1, \ldots, x_n\texttt{]}, ys) & = & \texttt{[}x_n, \ldots, x_1\texttt{]} \texttt{@} ys \end{array}$$

We have:

$$\begin{array}{lcl} n! & = & \texttt{factA}(n, 1) \\ \texttt{rev [}x_1, \ldots, x_n\texttt{]} & = & \texttt{revA}(\texttt{[}x_1, \ldots, x_n\texttt{]}, \texttt{[ ]}) \end{array}$$

m and *ys* are called *accumulating parameters*. They are used to hold the temporary result during the evaluation.

Declaration of `factA`

Property: `factA`$(n, m) = n! \cdot m$, for $n \geq 0$

```
let rec factA =
    function
    | (0,m) -> m
    | (n,m) -> factA(n-1,n*m) ;;
```

An evaluation:

$$
\begin{aligned}
& \texttt{factA(5,1)} \\
\rightsquigarrow\ & (\texttt{factA(n-1,n*m)}, [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow\ & \texttt{factA(4,5)} \\
\rightsquigarrow\ & (\texttt{factA(n-1,n*m)}, [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow\ & \texttt{factA(3,20)} \\
\rightsquigarrow\ & \ldots \\
\rightsquigarrow\ & \texttt{factA(0,120)} \rightsquigarrow (\texttt{m}, [\texttt{m} \mapsto 120]) \rightsquigarrow 120
\end{aligned}
$$

Space demand: constant.

Time demands: proportional to n

Declaration of `revA`

Property: `revA(`$[x_1, \ldots, x_n]$`,`*ys*`)` $=$ $[x_n, \ldots, x_1]$ `@`*ys*

```
let rec revA =
    function
    | ([], ys)     -> ys
    | (x::xs, ys) -> revA(xs, x::ys) ;;
```

An evaluation:

```
      revA([1,2,3],[])
⇝     revA([2,3],1::[])
⇝     revA([3],2::[1])
⇝     revA([3],[2,1])
⇝     revA([],3::[2,1])
⇝     revA([],[3,2,1])
⇝     [3,2,1]
```

Space and time demands:

proportional to *n* (the length of the first list)

Iterative (tail-recursive) functions (I)

The declarations of `factA` and `revA` are *tail-recursive functions*

- the recursive call is the *last function application* to be evaluated in the body of the declaration e.g. $\mathit{facA}(3, 20)$ and $\mathit{revA}([3], [2, 1])$
- only *one set* of bindings for argument identifiers is needed during the evaluation

Example

```
let rec factA =
    function
    | (0,m) -> m
    | (n,m) -> factA(n-1,n*m)
               (* recursive "tail-call" *)
```

- only one set of bindings for argument identifiers is needed during the evaluation

$$
\begin{aligned}
& \texttt{factA(5,1)} \\
\rightsquigarrow\ & (\texttt{factA(n,m)}, [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow\ & (\texttt{factA(n-1,n*m)}, [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow\ & \texttt{factA(4,5)} \\
\rightsquigarrow\ & (\texttt{factA(n,m)}, [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow\ & (\texttt{factA(n-1,n*m)}, [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow\ & \ldots \\
\rightsquigarrow\ & \texttt{factA(0,120)} \rightsquigarrow (\texttt{m}, [\texttt{m} \mapsto 120]) \rightsquigarrow 120
\end{aligned}
$$

Concrete resource measurements: factorial functions

```
let xs16 = List.init 1000000 (fun i -> 16);;
val xs16 : int list = [16; 16; 16; 16; 16; ...]

#time;; // a toggle in the interactive environment

for i in xs16 do let _ = fact i in ();;
Real: 00:00:00.051, CPU: 00:00:00.046, ...

for i in xs16 do let _ = factA(i,1) in ();;
Real: 00:00:00.024, CPU: 00:00:00.031, ...
```

The performance gain of `factA` is much better than the indicated factor 2 because the `for` construct alone uses about 12 ms:

```
for i in xs16 do let _ = () in ();;
Real: 00:00:00.012, CPU: 00:00:00.015, ...
```

Real: time elapsed by the execution. CPU: time spent by all cores.

Concrete resource measurements: reverse functions

```
let xs20000 = [1 .. 20000];;

naiveRev xs20000;;
Real: 00:00:07.624, CPU: 00:00:07.597,
GC gen0: 825, gen1: 253, gen2: 0
val it : int list = [20000; 19999; 19998; ...]

revA(xs20000,[]);;
Real: 00:00:00.001, CPU: 00:00:00.000,
GC gen0: 0, gen1: 0, gen2: 0
val it : int list = [20000; 19999; 19998; ...]
```

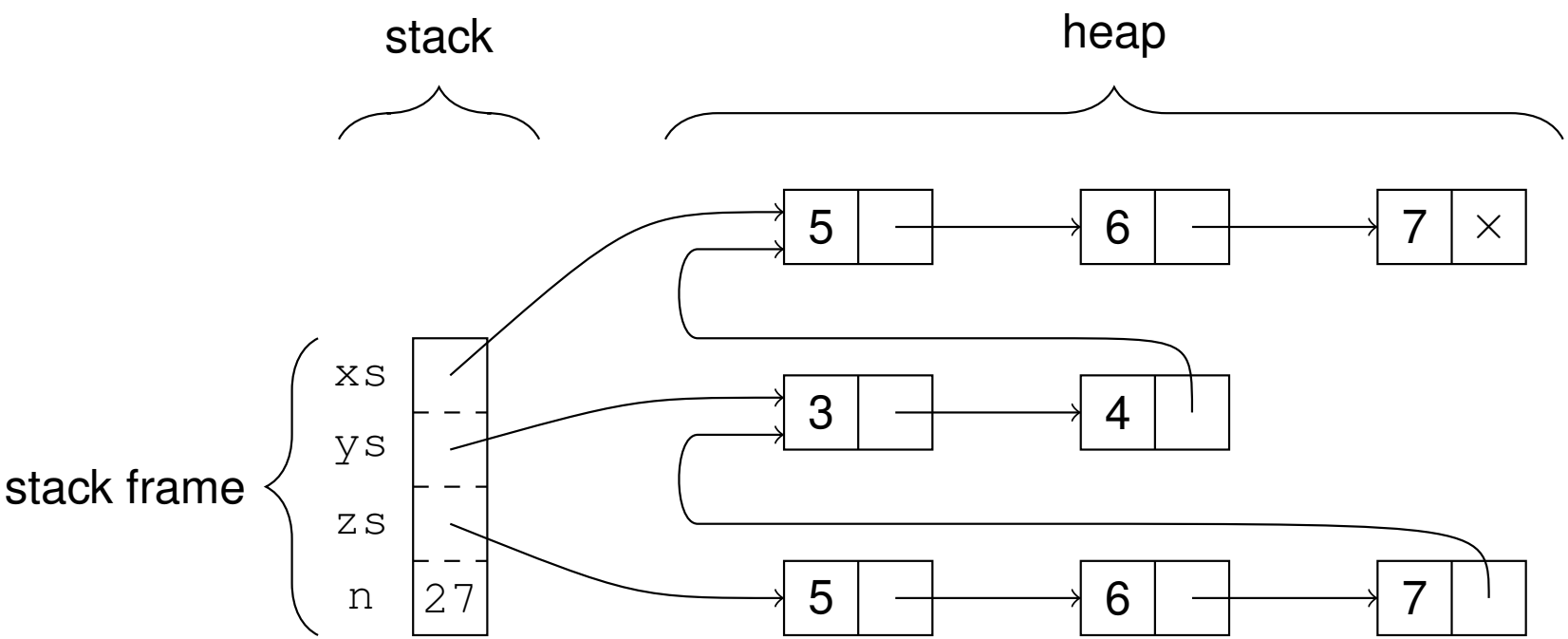
- The naive version takes 7.624 seconds - the iterative just 1 ms.
- The use of append (`@`) has been reduced to a use of cons (`::`). This has a dramatic effect of the garbage collection:
  - No object is reclaimed when `revA` is used
  - 825+253 obsolete objects were reclaimed using the naive version

Let's look at memory management

Memory management: stack and heap

- Primitive values are allocated on the stack
- Composite values are allocated on the heap

```
let xs = [5;6;7];;
let ys = 3::4::xs;;
let zs = xs @ ys;;
let n = 27;;
```



Observations

No unnecessary copying is done:

1. The linked lists for *ys* is not copied when building a linked list for *y* :: *ys*.
2. Fresh cons cells are made for the elements of *xs* only when building a linked list for *xs* `@` *ys*.

since a list is a functional (immutable) data structure

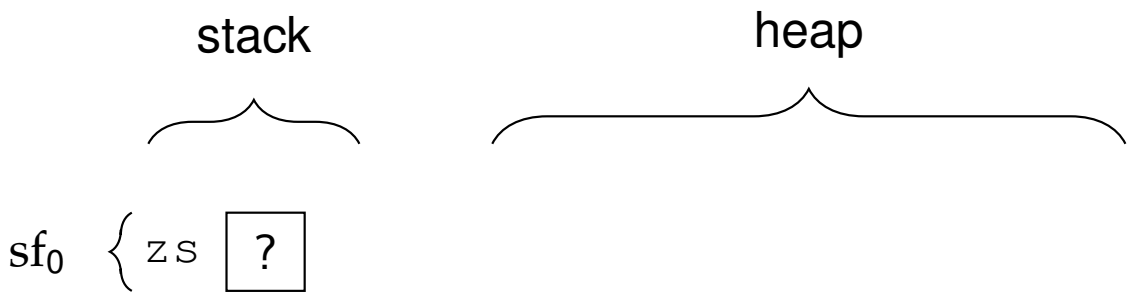
The running time of `@` is linear in the length of its first argument.

Operations on stack and heap

Example:

```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

Initial stack and heap prior to the evaluation of the local declarations:

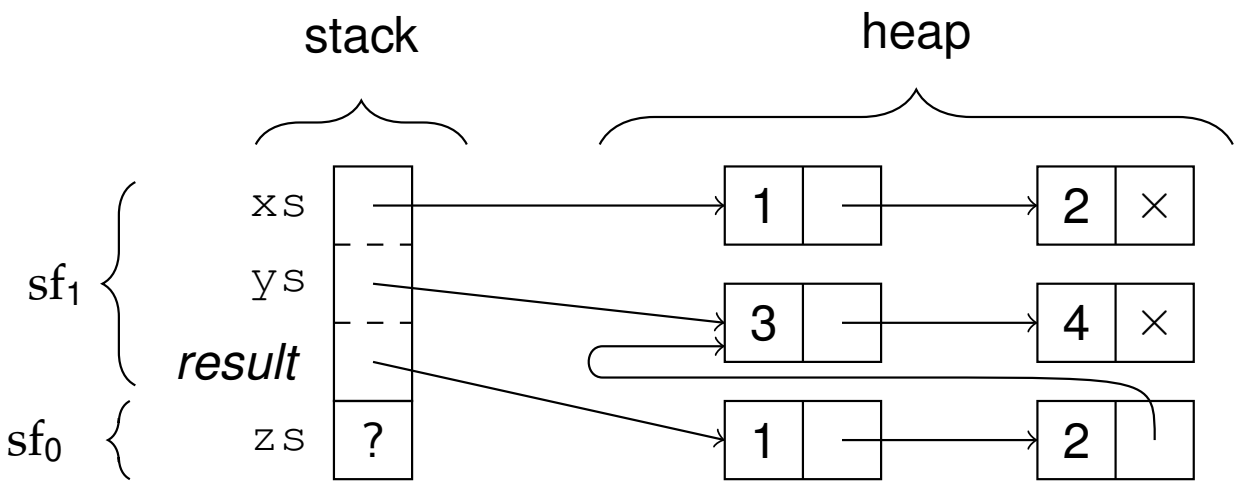


Operations on stack: Push

Example:

```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

Evaluation of the local declarations initiated by pushing a new stack frame onto the stack:



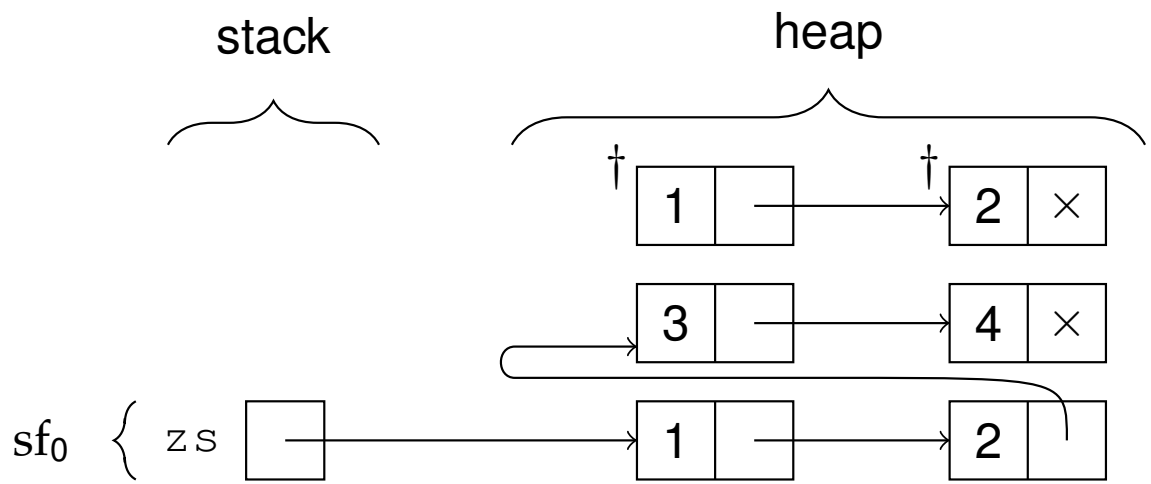
The auxiliary entry result refers to the value of the `let`-expression.

Operations on stack: Pop

Example:

```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

The top stack frame is popped from the stack when the evaluation of the `let`-expression is completed:



The resulting heap contains two obsolete cells marked with '†'

Operations on the heap: Garbage collection

The memory management system uses a *garbage collector* to reclaim obsolete cells in the heap behind the scene.

The garbage collector manages the heap as partitioned into three groups or *generations*: `gen0`, `gen1` and `gen2`, according to their age. The objects in `gen0` are the youngest while the objects in `gen2` are the oldest.

The typical situation is that objects die young and the garbage collector is designed for that situation.

Example:

```
naiveRev xs20000;;
Real: 00:00:07.624, CPU: 00:00:07.597,
GC gen0: 825, gen1: 253, gen2: 0
val it : int list = [20000; 19999; 19998; ...]
```

The limits of the stack and the heap

The stack is big:

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
bigList 120000;;
val it : int list = [1; 1; 1; 1; 1; 1; 1; 1;...]
bigList 130000;;
Process is terminated due to StackOverflowException.
```

More than $1.2 \cdot 10^5$ stack frames are pushed in recursive calls.

The heap is much bigger:

```
let rec bigListA n xs = if n=0 then xs
                        else bigListA (n-1) (1::xs);;
let xsVeryBig = bigListA 12000000 [];;
val xsVeryBig : int list = [1; 1; 1; 1; 1; 1;...]
let xsTooBig = bigListA 13000000 [];;
System.OutOfMemoryException: ...
```

A list with more than $1.2 \cdot 10^7$ elements can be created.

The iterative `bigListA` function does not exhaust the stack. WHY?

Iterative (tail-recursive) functions (II)

Tail-recursive functions are also called *iterative functions*.

- The function $f(n, m) = (n - 1, n * m)$ is iterated during evaluations for `factA`.
- The function $g(x :: xs, ys) = (xs, x :: ys)$ is iterated during evaluations for `revA`.

The correspondence between tail-recursive functions and while loops is established in the textbook.

An example:

```
let factW n =
    let ni = ref n
    let r  = ref 1
    while !ni>0 do
         r := !r * !ni ; ni := !ni-1
    !r;;
```

Iteration vs While loops

Iterative functions are executed efficiently:

```
#time;;

for i in 1 .. 1000000 do let _ = factA(16,1) in ();;
Real: 00:00:00.024, CPU: 00:00:00.031,
GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()

for i in 1 .. 1000000 do let _ = factW 16 in ();;
Real: 00:00:00.048, CPU: 00:00:00.046,
GC gen0: 9, gen1: 0, gen2: 0
val it : unit = ()
```

- the tail-recursive function actually is faster than the imperative while-loop based version

Iterative functions (III)

A function $g : \tau \texttt{ -> } \tau'$ is an *iteration of* $f : \tau \texttt{ -> } \tau$ if it is an instance of:

```
let rec g z = if p z then g(f z) else h z
```

for suitable predicate $p : \tau \texttt{ -> bool}$ and function $h : \tau \texttt{ -> } \tau'$.

The function g is called an *iterative (or tail-recursive) function*.

Examples: `factA` and `revA` are easily declared in the above form:

```
let rec factA(n,m) =
   if n<>0 then factA(n-1,n*m) else m;;
```

```
let rec revA(xs,ys) =
     if not (List.isEmpty xs)
     then revA(List.tail xs, (List.head xs)::ys)
     else ys;;
```

Iterative functions: evaluations (I)

Consider: `let rec g z = if p z then g(f z) else h z`

Evaluation of the $g\ v$:

$$
\begin{array}{ll}
 & g\ v \\
\rightsquigarrow & (\texttt{if}\ p\ \texttt{z}\ \texttt{then}\ g\,(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}\ ,\ [\texttt{z} \mapsto v]) \\
\rightsquigarrow & (g(f\ \texttt{z}),\ [\texttt{z} \mapsto v]) \\
\rightsquigarrow & g(f^1\,v) \\
\rightsquigarrow & (\texttt{if}\ p\ \texttt{z}\ \texttt{then}\ g\,(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}\ ,\ [\texttt{z} \mapsto f^1\,v]) \\
\rightsquigarrow & (g(f\ \texttt{z}),\ [\texttt{z} \mapsto f^1\,v]) \\
\rightsquigarrow & g(f^2\,v) \\
\rightsquigarrow & \ldots \\
\rightsquigarrow & (\texttt{if}\ p\ \texttt{z}\ \texttt{then}\ g\,(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}\ ,\ [\texttt{z} \mapsto f^n\,v]) \\
\rightsquigarrow & (h\ \texttt{z},\ [\texttt{z} \mapsto f^n\,v]) \qquad \text{suppose } p(f^n\,v) \rightsquigarrow \text{false} \\
\rightsquigarrow & h(f^n\,v)
\end{array}
$$

Iterative functions: evaluations (II)

Observe two desirable properties:

- there are n recursive calls of g,
- at most *one binding* for the argument pattern `z` is 'active' at any stage in the evaluation, and
- the iterative functions require one stack frame only.

Summary

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, for example, in order
  - to avoid evaluations with a huge amount of pending operations, e.g.

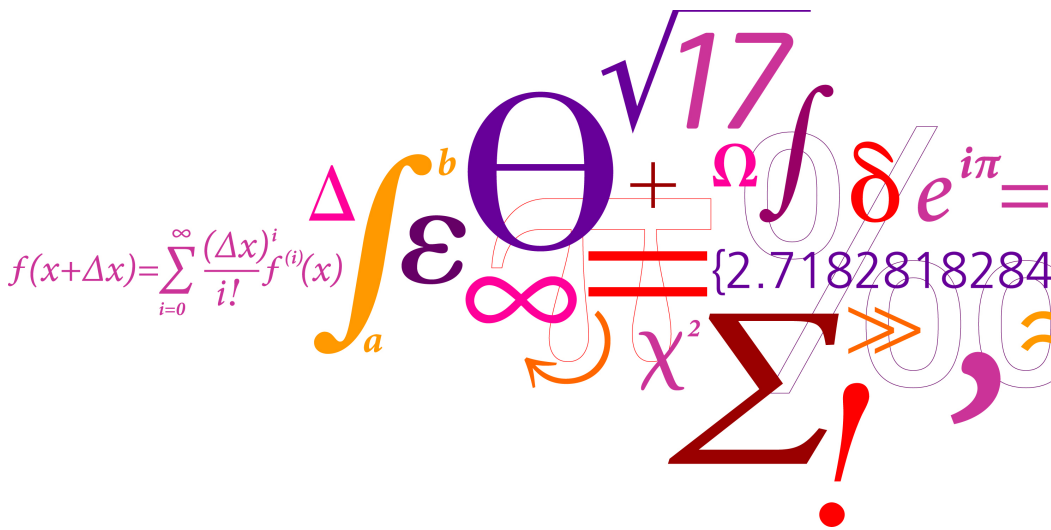
$$7+(6+(5\cdots+f\ 2\cdots))$$

  - to avoid inadequate use of `@` in recursive declarations.
- Iterative functions with accumulating parameters correspond to while-loops

02157 Functional Programming

Lecture 9: Tail-recursive (iterative) functions (II)

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

Last week:

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, for example, in order
  - to avoid evaluations with a huge amount of pending operations, e.g.

$$7+(6+(5\cdots+f\ 2\cdots))$$

  - to avoid inadequate use of `@` in recursive declarations.
- Iterative functions with accumulating parameters correspond to while-loops

Today:

- The concept continuation: Provides a general applicable approach to achieving tail-recursive functions.
  - Can also cope with complicated control structures, such as backtracking and exception handling.
  - Recommended supplementary reading: Chapters 11 and 12: *Programming Language Concepts*, Peter Sestoft, Springer 2012.

Limitation of accumulating parameters

A tail-recursive version of a recursive functions CANNOT be obtained using an accumulating parameter in all cases.

Consider for example:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a>*'a*BinTree<'a>;;

let rec count = function
                | Leaf           -> 0
                | Node(tl,n,tr) -> count tl + count tr + 1;
```

A counting function:

```
countA: int -> BinTree<'a> -> int
```

using an accumulating parameter will not be tail-recursive due to the expression containing recursive calls on the left and right sub-trees. (Ex. 9.8)

Continuations

Continuation: A representation of the “rest” of the computation.

Example: A summation function that is NOT tail-recursive: WHY?

```
let rec sum xs = match xs with
                 | []     -> 0
                 | x::rst -> x + sum rst;;
```

The continuation-based version of `sum` has

```
k: int -> int
```

as an extra argument. Determines what happens with the result.

```
let rec sumC xs k = match xs with
                    | []     -> k 0
                    | x::rst -> sumC rst (fun v -> k(x+v))
```

This is a tail-recursive function.

- Base case: “feed” the result into the continuation `k`.
- Recursive case: The continuation after `rst` is processed is a function of the result `v` that
  - performs the addition `x+v` and
  - feed that result into the continuation `k`

An evaluation

```
    sumC [1;2;3] id
⇝   sumC [2;3] (fun v -> id(1+v))
⇝   sumC [3] (fun w -> (fun v -> id(1+v))(2+w))
⇝   sumC [] (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u))
⇝   (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u)) 0
⇝   (fun w -> (fun v -> id(1+v))(2+w)) 3
⇝   (fun v -> id(1+v)) 5
⇝   id 6
⇝   6
```

Notice:

- Closures are allocated in the heap.
- Just one stack frame is needed due to tail calls.
- Stack is traded for heap.

A more efficient representation of the continuation

Consider the following version using an accumulating parameter:

```
let rec sumA xs n = match xs with
                    | []     -> n
                    | x::rst -> sumA rst (x+n);;
```

Remember:

```
let rec sumC xs k = match xs with
                    | []     -> k 0
                    | x::rst -> sumC rst (fun v -> k(x+v))
```

What is the relationship between `sumA` and `sumC`?

$$\texttt{sumA}\ xs\ n = \texttt{sumC}\ xs\ (\texttt{fun}\ v \rightarrow n + v)$$

Proof: Structural induction over lists.

Tail recursion using accumulating parameters is not always achievable.

Structural induction over lists

The declaration

```
type 'a list =
   | Nil                        //        Nil is written []
   | Cons of 'a * 'a list       // Cons(x,xs) is written x::xs
```

denotes an inductive definition of lists (of type 'a)

- [] is a list
- if x is an element and xs is a list, then $x :: xs$ is a list
- lists can be generated by above rules only

The following structural induction rule is therefore sound:

$$\frac{\begin{array}{lll} 1. & P([]) & \text{base case} \\ 2. & \forall xs.\forall x.(P(xs) \Rightarrow P(x :: xs)) & \text{inductive step} \end{array}}{\forall xs.P(xs)}$$

A simple verification

```
let rec sumA xs n = match xs with
                    | []    -> n
                    | x::rst -> sumA rst (x+n)

let rec sumC xs k = match xs with
                    | []    -> k 0
                    | x::rst -> sumC rst (fun v -> k(x+v))
```

Property: `sumA` *xs n* = `sumC` *xs* (`fun` *v* → *n* + *v*).

Proof: Structural induction over lists. Base case *xs* = [] is easy.

$\forall n'.$`sumA` *xs* n' = `sumC` *xs* (`fun` *v* → n' + *v*) Ind. Hyp.

The inductive step is proved as follows:

```
  sumC (x::xs) (fun v -> n+v)
=sumC xs (fun w -> ((fun v -> n+v)(x+w)))
=sumC xs (fun w -> n+(x+w))                beta reduction
=sumC xs (fun w -> (n+x)+w)                arithmetic
=sumA xs (n+x)                             ind. hyp. with n' ↦ n + x
=sumA (x::xs) n
```

Continuations: Another example

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
```

The continuation-based version of `bigList` has a continuation

```
k: int list -> int list
```

as argument:

```
let rec bigListC n k =
   if n=0 then k []
   else bigListC (n-1) (fun res -> k(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

- Base case: "feed" the result into the continuation `k`.
- Recursive case: The continuation after processing `(n-1)` is the function of the result `res` that
  - builds the list `1::res` and
  - feeds that list into the continuation `k`.

Observations

- `bigListC` is a tail-recursive function, and
- the calls of `k` are tail calls in the base case of `bigListC` and in the continuation: `fun res -> k(1::res)`.

The stack will hence neither grow due to the evaluation of recursive calls of `bigListC` nor due to calls of the continuations that have been built in the heap:

```
bigListC 16000000 id;;
Real: 00:00:08.586, CPU: 00:00:08.314,
GC gen0: 80, gen1: 60, gen2: 3
val it : int list = [1; 1; 1; 1; 1;...]
```

- Slower than `bigList`
- Can generate longer lists than `bigList`

Example: Tail-recursive count

```
let rec countC t k =
  match t with
  | Leaf           -> k 0
  | Node(tl,n,tr) ->
     countC tl (fun vl -> countC tr (fun vr -> k(vl+vr+1)))
val countC : BinTree<'a> -> (int -> 'b) -> 'b

countC (Node(Node(Leaf,1,Leaf),2,Node(Leaf,3,Leaf))) id;;
val it : int = 3
```

- Both calls of `countC` are tail calls
- The two calls of `k` are tail calls

Hence, the stack will not grow when evaluating `countC` *t k*.

- `countC` can handle bigger trees than `count`
- `count` is faster

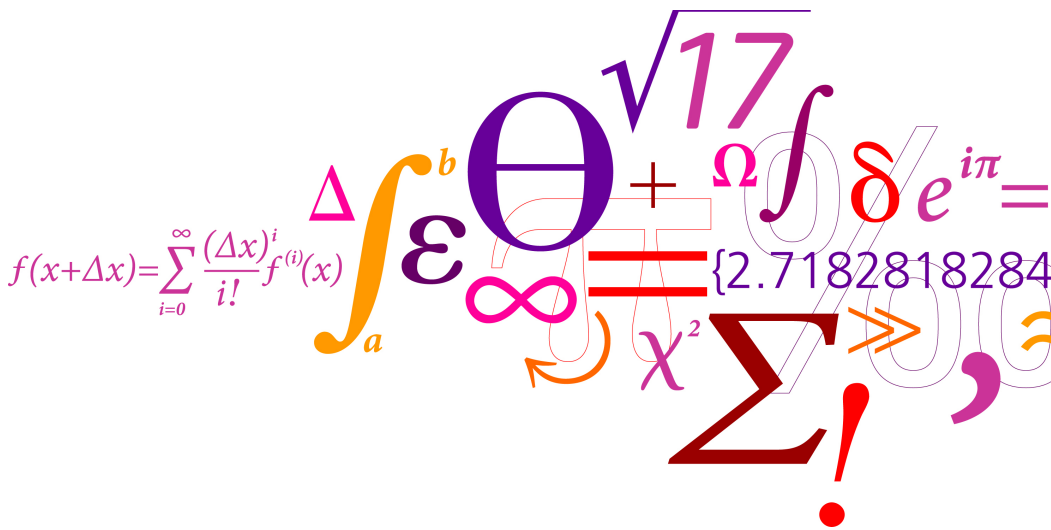
Summary and recommendations

- Have iterative functions in mind when dealing with efficiency, e.g.
  - to avoid evaluations with a huge amount of pending operations
  - to avoid inadequate use of `@` in recursive declarations.
- Memory management: stack, heap, garbage collection
- Continuations – provide a technique to turn arbitrary recursive functions into tail-recursive ones. trades stack for heap

02157 Functional Programming

Sequences and Sequence Expressions

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Sequence: a possibly infinite, ordered collection of elements, where the elements are computed by demand only

- the sequence concept
- standard sequence functions – the `Seq` library
- sequence expressions – computation expressions used generate sequences in a step by step manner

Computation expressions: provide a mean to express specific kinds of computations where low-level details are hidden. See Chapter 12.

Sequences (or Lazy Lists)

- *lazy evaluation* or *delayed evaluation* is the technique of delaying a computation until the result of the computation is needed.

  Default in lazy languages like Haskell

It is occasionally efficient to be lazy.

A special form of this is a *sequence*, where the elements are not evaluated until their values are required by the rest of the program.

- a *sequence* may be infinite

  just a finite part is used in computations

Example:

- Consider the sequence of all prime numbers: $2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots$
- the first 5 are $2, 3, 5, 7, 11$

Sieve of Eratosthenes

Delayed computations in eager languages

The computation of the value of *e* can be delayed by ”packing” it into a function (a closure):

```
fun () -> e
```

Example:

```
fun () -> 3+4;;
val it : unit -> int = <fun:clo@10-2>

it();;
val it : int = 7
```

The addition is deferred until the closure is applied.

How can we convince ourselves that the addition deferred?

Example continued

A use of side effects may reveal when computations are performed:

```
let idWithPrint i = let _ = printfn "%d" i
                    i;;
val idWithPrint : int -> int

idWithPrint 3;;
3
val it : int = 3
```

The value is printed before it is returned.

```
fun () -> (idWithPrint 3) + (idWithPrint 4);;
val it : unit -> int = <fun:clo@14-3>
```

Nothing is printed yet.

```
it();;
3
4
val it : int = 7
```

Sequences in F#

A lazy list or *sequence* in F# is a possibly infinite, ordered collection of elements, where the elements are computed by demand only.

The natural number sequence $0, 1, 2, \ldots$ is created as follows:

```
let nat = Seq.initInfinite id;;
val nat : seq<int>
```

where `id:'a->'a` is the built-in identity function, i.e. $\texttt{id}(x) = x$

No element in the sequence is generated yet!

The type `seq<'a>` is an abstract datatype.

Programs on sequences are constructed from `Seq`-library functions

Explicit sequences and conversions for finite sequences

Two conversion functions

```
Seq.toList: seq<'a> -> 'a list

Seq.ofList: 'a list -> seq<'a>
```

with examples

```
let sq = Seq.ofList ['a' .. 'f'];;
val sq : seq<char> = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']

let cs = Seq.toList sq;;
val cs : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
```

Alternatively, an finite sequence can written as follows:

```
let sq = seq ['a' .. 'f'];;
val sq : seq<char> = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
```

Notice

- Seq.toList – does not terminate for infinite sequences
- seq $[x_1; \ldots; x_n]$ is a finite sequence with $n \geq 0$ elements

Selected functions from the library: Seq

- `initInfinite: (int ->'a) -> seq<'a>`.
  `initInfinite` *f* generates the sequence $f(0), f(1), f(2), \ldots$
- `delay: (unit->seq<'a>) -> seq<'a>`.
  `delay` *g* generates the elements of *g*() lazily
- `collect: ('a->seq<'b>) -> seq<'a> -> seq<'b>`.
  `collect` *f sq* generates the sequence obtained by appending the sequences: $f(sq_0), f(sq_1), f(sq_2), \ldots$

The Seq library contains functions, e.g. `collect`, that are sequence variants of functions from the List library. Other examples are:

- `item: int -> seq<'a> -> 'a`
- `head: seq<'a> -> 'a`
- `tail: seq<'a> -> seq<'a>`
- `append: seq<'a> -> seq<'a> -> seq<'a>`
- `take: int -> seq<'a> -> seq<'a>`
- `filter: ('a->bool) -> seq<'a> -> seq<'b>`.

Example continued

A `nat` element is computed by demand only:

```
let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>
```

— using `idWithPrint` to inspect element generation.

Demanding an element of the sequence:

```
Seq.item 4 nat;;
4
val it : int = 4
```

Just the 5th element is generated

Further examples

A sequence of even natural numbers is easily obtained:

```
let even = Seq.filter (fun n -> n%2=0) nat;;
val even : seq<int>

Seq.toList(Seq.take 4 even);;
0
1
2
3
4
5
6
val it : int list = [0; 2; 4; 6]
```

Demanding the first 4 even numbers requires a computation of the first 7 natural numbers.

Sieve of Eratosthenes

Greek mathematician (194 – 176 BC)

Computation of prime numbers

- start with the sequence $2, 3, 4, 5, 6, \ldots$
  select head (2), and remove multiples of 2 from the sequence
  2
- next sequence $3, 5, 7, 9, 11, \ldots$
  select head (3), and remove multiples of 3 from the sequence
  $2, 3$
- next sequence $5, 7, 11, 13, 17, \ldots$
  select head (5), and remove multiples of 5 from the sequence
  $2, 3, 5$
- $\vdots$

Sieve of Eratosthenes in F# (I)

Remove multiples of *a* from sequence *sq*:

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq;;
val sift : int -> seq<int> -> seq<int>
```

Select head and remove multiples of head from the tail – recursively:

```
let rec sieve sq =
   Seq.delay (fun () ->
                  let p = Seq.head sq
                  Seq.append
                      (seq [p])
                      (sieve(sift p (Seq.tail sq))));;
val sieve : seq<int> -> seq<int>
```

- A delay is needed to avoid infinite recursion Why?

Sequence expressions support a more natural formulation

Examples

The sequence of prime numbers and the *n*'th prime number:

```
let primes = sieve(Seq.initInfinite (fun n -> n+2));;
val primes : seq<int>

let nthPrime n = Seq.item n primes;;
val nthPrime : int -> int

nthPrime 100;;
val it : int = 547
```

Re-computation can be avoided by using cached sequences:

```
let primesCached = Seq.cache primes;;

let nthPrime' n = Seq.item n primesCached;;
val nthPrime' : int -> int
```

Computing the 700'th prime number takes about 4.5s; a subsequent computation of the 705'th is fast since that computation starts from the 700 prime number

Sieve of Eratosthenes using Sequence Expressions

Sequence expressions can be used for defining sequences in a step-by-step generation manner.

The sieve of Erastothenes:

```
let rec sieve sq =
    seq { let p = Seq.head sq
          yield p
          yield! sieve(sift p (Seq.tail sq)) };;
val sieve : seq<int> -> seq<int>
```

- By construction lazy – no need to use `Seq.delay`
- `yield` *x* adds the element *x* to the generated sequence
- `yield!` *sq* adds the sequence *sq* to the generated sequence
- *seqexp*$_1$ *seqexp*$_2$ appends the sequences *seqexp*$_1$ and *seqexp*$_2$

Defining sift using Sequence Expressions

The sift function can be defined using an iteration:

`for` *pat* `in` *exp* `do` *seqexp*

and a filter:

`if` *exp* `then` *seqexp*

as follows:

```
let sift a sq = seq { for n in sq do
                        if n % a <> 0 then
                          yield n       };;
val sift : int -> seq<int> -> seq<int>
```

Example: Catalogue search (I)

Extract (recursively) the sequence of all files in a directory:

```
open System.IO ;;

let rec allFiles dir =
  seq {yield! Directory.GetFiles dir
       yield! Seq.collect allFiles (Directory.GetDirectories dir)}
val allFiles : string -> seq<string>
```

where

`Seq.collect: ('a -> seq<'c>) -> seq<'a> -> seq<'c>`

combines a 'map' and 'concatenate' functionality.

```
Directory.SetCurrentDirectory @"C:\mrh\Forskning\Cambridge\";;
let files = allFiles ".";;
val files : seq<string>

Seq.item 100 files;;
val it : string = ".\BOOK\Satisfiability.fs"
```

Nothing is computed beyond element 100.

Summary

- Anonymous functions `fun () ->` *e* can be used to delay the computation of *e*.
- Possibly infinite sequences provide natural and useful abstractions
- The computation by demand only is convenient in many applications

It is occasionally efficient to be lazy.

The type `seq<'a>` is a synonym for the .NET type `IEnumerable<'a>`.

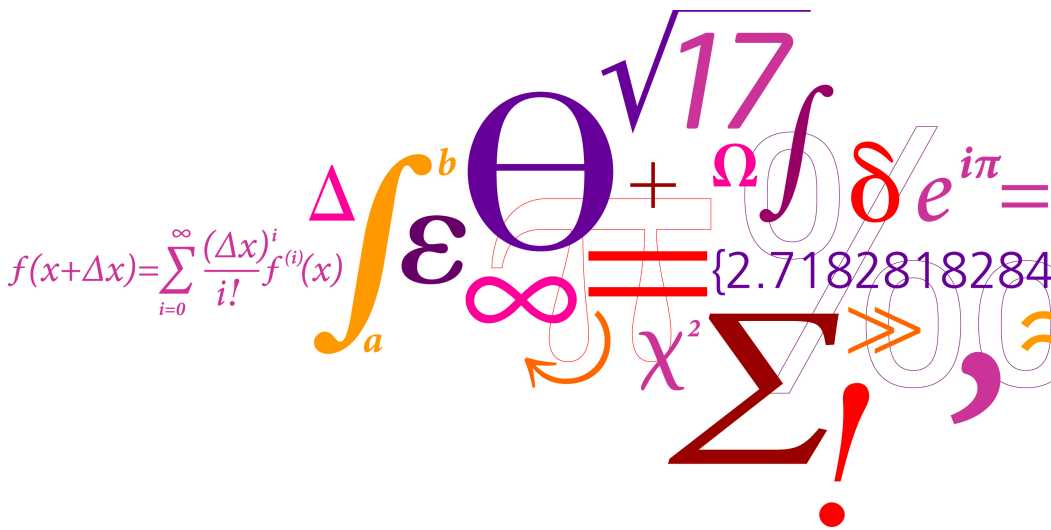
Any .NET type that implements this interface can be used as a sequence.

- Lists, arrays and databases, for example.

02157 Functional Programming

Lecture 11: Module System – briefly

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Supports modular program design including
  - encapsulation
  - abstraction and
  - reuse of software components.
- A module is characterized by:
  - a *signature* – an interface specifications and
  - a matching *implementation* – containing declarations of the interface specifications.
- Example based (incomplete – no object interface types, for example) presentation to give the flavor.

Sources:

- Chapter 7: Modules. (A fast reading suffices.)

An example: Search trees

Consider the following implementation of search trees:

```
type Tree =   Lf
            | Br of Tree*int*Tree;;

let rec insert i = function
  | Lf                  ->  Br(Lf,i,Lf)
  | Br(t1,j,t2) as tr ->
      match compare i j with
       | 0             -> tr
       | n when n<0    -> Br(insert i t1 , j, t2)
       | _             -> Br(t1,j, insert i t2);;


let rec memberOf  i = function
  | Lf           -> false
  | Br(t1,j,t2) -> match compare i j with
                   | 0   -> true
                   | n when n<0 -> memberOf i t1
                   | _          -> memberOf i t2;;
```

Example cont'd

Is this implementation adequate?

No. Search tree property can be violated by a programmer:

```
toList(insert 2 (Br(Br(Lf,3,Lf), 1, Br(Lf,0,Lf))));;
val it = [3;1;0;2]: int list
```

Solution: Hide the internal structure of search trees.

Module

A module is a combination of a

- signature, which is a specification of an interface to the module (the user's view), and an
- implementation, which provides declarations for the specifications in the signature.

Geometric vectors: Signature

The signature specifies one type and eight values:

```
// Vector signature
module Vector
type vector
val ( ~-. ) : vector -> vector              // Vector sign change
val ( +. )  : vector -> vector -> vector    // Vector sum
val ( -. )  : vector -> vector -> vector    // Vector difference
val ( *. )  : float  -> vector -> vector    // Product with number
val ( &. )  : vector -> vector -> float     // Dot product
val norm    : vector -> float               // Length of vector
val make    : float * float -> vector       // Make vector
val coord   : vector -> float * float       // Get coordinates
```

The specification 'vector' does not reveal the implementation

- Why is make and coord introduced?

Geometric vectors (2): Simple implementation

An implementation must declare each specification of the signature:

```
// Vector implementation
module Vector
type vector = V of float * float
let (~-.) (V(x,y))                = V(-x,-y)
let (+.) (V(x1,y1)) (V(x2,y2)) = V(x1+x2,y1+y2)
let (-.)  v1          v2          = v1 +. -. v2
let ( *.) a          (V(x1,y1)) = V(a*x1,a*y1)
let (&.) (V(x1,y1)) (V(x2,y2)) = x1*x2 + y1*y2
let norm  (V(x1,y1))              = sqrt(x1*x1+y1*y1)
let make  (x,y)                   = V(x,y)
let coord (V(x,y))                = (x,y)
```

- Since the representation of 'vector' is hidden in the signature, the type must be implemented by either a tagged value or a record.

Geometric vectors (3): Compilation

Suppose

- the signature is in a file 'Vector.fsi'
- the implementation is in a file 'Vector.fs'

A library file 'Vector.dll' is constructed by the following command:

```
D:\MRH data\ ... \Libraries\fsc -a Vector.fsi Vector.fs
```

The library 'Vector' can now be used just like other libraries, such as 'Set' or 'Map'.

- Compiler on Linux and Mac systems: fsharpc

Geometric vectors (4): Use of library

A library must be referenced before it can be used.

```
 #r @"d:\MRH data\ ... \Libraries\Vector.dll";;
--> Referenced 'd:\MRH data\ ... \Libraries\Vector.dll'
open Vector ;;

let a = make(1.0,-2.0);;
val a : vector
let b = make(3.0,4.0);;
val b : vector
let c = 2.0 *. a -. b;;
val c : vector

coord c ;;
val it : float * float = (-1.0, -8.0)

let d = c &. a;;
val d : float = 15.0

let e = norm b;;
val e : float = 5.0
```

Notice: the implementation of `vector` is not visible and it cannot be exploited.

Type augmentation

A *type augmentation*

- adds declarations to the definition of a tagged type or a record type
- allows declaration of (overloaded) operators.

In the 'Vector' module we would like to

- overload $+$, $-$ and $\star$ to also denote vector operations.
- overload $\star$ to denote two different operations on vectors.

Type augmentation – signature

```
module Vector

[<Sealed>]
type vector =
  static member ( ~- ) : vector -> vector
  static member ( + )  : vector * vector -> vector
  static member ( - )  : vector * vector -> vector
  static member ( * )  : float  * vector -> vector
  static member ( * )  : vector * vector -> float
val make : float * float -> vector
val coord: vector -> float * float
val norm : vector -> float
```

- The *attribute* `[<Sealed>]` is mandatory when a type augmentation is used.
- The "`member`" specification and declaration of an infix operator (e.g. `+`) correspond to a type of form $type_1$ `*` $type_2$ `->` $type_3$
- The operators can still be used on numbers.

Type augmentation – implementation and use

```
module Vector

type vector =
  | V of float * float
  static member (~-) (V(x,y))             = V(-x,-y)
  static member (+) (V(x1,y1),V(x2,y2)) = V(x1+x2,y1+y2)
  static member (-) (V(x1,y1),V(x2,y2)) = V(x1-x2,y1-y2)
  static member (*) (a, V(x,y))           = V(a*x,a*y)
  static member (*) (V(x1,y1),V(x2,y2)) = x1*x2 + y1*y2
let make (x,y)      = V(x,y)
let coord (V(x,y)) = (x,y)
let norm (V(x,y))   = sqrt(x*x + y*y)
```

The operators $+$, $-$, $*$ are available on vectors even without opening:

```
let a = Vector.make(1.0,-2.0);;
val a : Vector.vector

let b = Vector.make(3.0,4.0);;
val b : Vector.vector

let c = 2.0 * a - b;;
val c : Vector.vector
```

Customizing the string function

```
module Vector
type vector =
   | V of float * float
   override v.ToString() =
            match v with | V(x,y) -> string(x,y)

let make (x,y)      = V(x,y)
   ...
type vector with
  static member (~-) (V(x,y))            = V(-x,-y)
  ...
```

- The default ToString function that do not reveal a meaningful value is overridden to give a string for the pair of coordinates.
- A type extension is used.

Example:

```
let a = Vector.make(1.0,2.0);;
val a : Vector.vector = (1, 2)

string(a+a);;
val it : string = "(2, 4)"
```

Summary

Modular program development

- program libraries using signatures and structures
- type augmentation, overloaded operators, customizing string (and other) functions
- Encapsulation, abstraction, reuse of components, division of concerns, ...
- ...