

## 02157 Functional Programming

## Lecture 7: Module System – briefly

Michael R. Hansen

$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$

DTU Compute

Department of Applied Mathematics and Computer Science

- On critical looks at programs
  - simplification for readability reasons
  - simplifications for computational reasons
- The Module System
- F# is integrated on .Net
  - free, open source
  - cross platform for Linux, MacOS, Windows, ...
  - A three project solution for polynomials containing
    - An F# library (class library)
    - An F# console application
    - A C# console application
- A brief look at type inference

briefly

on Learn

## On critical looks at programs

## On critical looks at programs

- Aim at succinct programs

## On critical looks at programs

- Aim at succinct programs  
have a critical look at your own programs

## On critical looks at programs

- Aim at succinct programs  
have a critical look at your own programs
- Correctness has top priority

## On critical looks at programs

- Aim at succinct programs  
have a critical look at your own programs
- Correctness has top priority  
but have an eye to sensible use of resources

# Can we simplify?

Have a look at:

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



# Can we simplify?

Have a look at:

```
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?

# Can we simplify?

Have a look at:

```
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?

Can we improve readability?

# Is a use of library functions adequate?



Have a look at

```
let h a xs = let xs1 = List.filter (fun (a',t) -> a=a') xs
              let xs2 = List.map (fun (a,t:int) -> t) xs1
              List.sum xs2
```

# Is a use of library functions adequate?

Have a look at

```
let h a xs = let xs1 = List.filter (fun (a',t) -> a=a') xs
              let xs2 = List.map (fun (a,t:int) -> t) xs1
              List.sum xs2
```

- Can this problem easily be solved in a less resource demanding manner?

Have a look at:

```
let rec h1 a =  
  function  
  | []                                -> 0  
  | (a',t)::rest when a=a'          -> t + h1 a rest  
  | _::rest                          -> h1 a rest;;  
  
let rec h2 a xs =  
  List.fold (fun s (a',t) -> if a=a' then s+t else s) 0 xs;
```

Have a look at:

```
let rec h1 a =  
  function  
  | []                -> 0  
  | (a',t)::rest when a=a' -> t + h1 a rest  
  | _::rest          -> h1 a rest;;  
  
let rec h2 a xs =  
  List.fold (fun s (a',t) -> if a=a' then s+t else s) 0 xs;
```

Solutions are based on a simple algorithmic idea

- traverse the list `xs` one time and
- build up the result during the traversal

Have a look at:

```
let rec h1 a =  
  function  
  | []                                -> 0  
  | (a',t)::rest when a=a' -> t + h1 a rest  
  | _::rest                      -> h1 a rest;;  
  
let rec h2 a xs =  
  List.fold (fun s (a',t) -> if a=a' then s+t else s) 0 xs;
```

Solutions are based on a simple algorithmic idea

- traverse the list `xs` one time and
- build up the result during the traversal

Correctness has top priority

Have a look at:

```
let rec h1 a =  
  function  
  | []                -> 0  
  | (a',t)::rest when a=a' -> t + h1 a rest  
  | _::rest          -> h1 a rest;;
```

```
let rec h2 a xs =  
  List.fold (fun s (a',t) -> if a=a' then s+t else s) 0 xs;
```

Solutions are based on a simple algorithmic idea

- traverse the list `xs` one time and
- build up the result during the traversal

Correctness has top priority

– but have an eye to sensible use of resources



Have a look at:

```
let rec h1 a =  
  function  
  | []                -> 0  
  | (a',t)::rest when a=a' -> t + h1 a rest  
  | _::rest          -> h1 a rest;;
```

```
let rec h2 a xs =  
  List.fold (fun s (a',t) -> if a=a' then s+t else s) 0 xs;
```

Solutions are based on a simple algorithmic idea

- traverse the list `xs` one time and
- build up the result during the traversal

Correctness has top priority

– but have an eye to sensible use of resources

More when the topic: Tail recursion, is covered

## The module system

- Supports modular program design including
  - encapsulation
  - abstraction and
  - reuse of software components.

- 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.

- 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 presentation to give the flavor  
incomplete – no object interface types, for example

- 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 presentation to give the flavor  
incomplete – no object interface types, for example

## Sources:

- Chapter 7: Modules. (A fast reading suffices.)

## On hiding: Polynomial program

An violation of a representation invariant

without hiding

```
// A misuse: [0] is not legal polynomial
let p1 = mulX [0];;
// val p1 : int list = [0; 0]

// Pretty print
toString p1;;
// val it : string = "0"
```

## On hiding: Polynomial program

An violation of a representation invariant

without hiding

```
// A misuse: [0] is not legal polynomial
let p1 = mulX [0];;
// val p1 : int list = [0; 0]

// Pretty print
toString p1;;
// val it : string = "0"

let p2 = mulX [];;
// val p2 : int list = []

// Pretty print
toString p2;;
// val it : string = "0"
```



## On hiding: Polynomial program

An violation of a representation invariant

without hiding

```
// A misuse: [0] is not legal polynomial
let p1 = mulX [0];;
// val p1 : int list = [0; 0]

// Pretty print
toString p1;;
// val it : string = "0"

let p2 = mulX [];;
// val p2 : int list = []

// Pretty print
toString p2;;
// val it : string = "0"

// But
p1 = p2;;
// val it : bool = false
```

- may cause unpredictable results

```
#r @"Polynomial.dll"

open Polynomial

let p1 = mulX (ofList [0]);;
// val p1 : Poly = 0           pretty print

let p2 = mulX (ofList []);;
// val p2 : Poly = 0          pretty print

p1 = p2;;
// val it : bool = true
```

```
#r @"Polynomial.dll"

open Polynomial

let p1 = mulX (ofList [0]);;
// val p1 : Poly = 0           pretty print

let p2 = mulX (ofList []);;
// val p2 : Poly = 0          pretty print

p1 = p2;;
// val it : bool = true
```

- internal representation is hidden

```
#r @"Polynomial.dll"

open Polynomial

let p1 = mulX (ofList [0]);;
// val p1 : Poly = 0           pretty print

let p2 = mulX (ofList []);;
// val p2 : Poly = 0          pretty print

p1 = p2;;
// val it : bool = true
```

- internal representation is hidden
- `ofList` gives legal representations

```
#r @"Polynomial.dll"

open Polynomial

let p1 = mulX (ofList [0]);;
// val p1 : Poly = 0           pretty print

let p2 = mulX (ofList []);;
// val p2 : Poly = 0           pretty print

p1 = p2;;
// val it : bool = true
```

- internal representation is hidden
- `ofList` gives legal representations
- functions preserve the invariant: `isLegal`

## Protection of a representation invariant

```
#r @"Polynomial.dll"

open Polynomial

let p1 = mulX (ofList [0]);;
// val p1 : Poly = 0           pretty print

let p2 = mulX (ofList []);;
// val p2 : Poly = 0          pretty print

p1 = p2;;
// val it : bool = true
```

- internal representation is hidden
- `ofList` gives legal representations
- functions preserve the invariant: `isLegal`

Unpredictable results are prevented

A **module** is a combination of a

- **signature**, which is a specification of an interface to the module (the user's view), and an

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.



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 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

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?

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)
```

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**'.

## 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 (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**

An alternative is to use the **Command Line Interface (CLI)** tool  
mentioned later in this lecture



## 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.

A *type augmentation*

- adds declarations to the definition of a tagged type or a record type
- allows declaration of (overloaded) operators.

## 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  $*$  to also denote *vector* operations.
- overload  $*$  to denote *two* different operations on vectors.

```
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
```

## 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.

# 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  $type_1 * type_2 \rightarrow type_3$

# 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  $type_1 * type_2 \rightarrow type_3$
- 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)
```



# 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 does not reveal a meaningful value is overridden to give a string for the pair of coordinates.
- A type extension is used.

# 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 does 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)"

```

## 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, ...
- ...

# A three project solution for polynomials containing on Learn

# Nano-solution to Polynomial exercise on Learn

A **three-project solution** to a minimal part of the polynomial exercise:

- **PolyLib**: an F# library for polynomials a class library
- **CSharpApp** a C# console application using the library
- **FSharpApp** an F# console application using the library

# Nano-solution to Polynomial exercise on Learn

A **three-project solution** to a minimal part of the polynomial exercise:

- **PolyLib**: an F# library for polynomials a class library
- **CSharpApp** a C# console application using the library
- **FSharpApp** an F# console application using the library

The solution is formed using the **Command Line Interface (CLI)** tool

- `https://docs.microsoft.com/en-us/dotnet/fsharp/get-started/get-started-command-line`
- `https://docs.microsoft.com/en-us/dotnet/core/tools`  
works for Linux, MacOS, Windows operating systems

# Nano-solution to Polynomial exercise on Learn

A **three-project solution** to a minimal part of the polynomial exercise:

- **PolyLib**: an F# library for polynomials a class library
- **CSharpApp** a C# console application using the library
- **FSharpApp** an F# console application using the library

The solution is formed using the **Command Line Interface (CLI)** tool

- `https://docs.microsoft.com/en-us/dotnet/fsharp/get-started/get-started-command-line`
- `https://docs.microsoft.com/en-us/dotnet/core/tools`  
**works for Linux, MacOS, Windows operating systems**

Consult DTU Learn concerning:

- How the solution is formed using the CLI tools
- How to run the applications and the script
- How executables and libraries (assemblies / binaries) are build



## Solution with three projects

- PolyLib

F# class library

- PolyLib.fsproj includes compilation information
- Polynomial.fsi signature (interface) file
- Polynomial.fs implementation file
- script.fsx reference to PolyLib.dll

# Overview of Solution

## Solution with three projects

- PolyLib F# class library
  - PolyLib.fsproj includes compilation information
  - Polynomial.fsi signature (interface) file
  - Polynomial.fs implementation file
  - script.fsx reference to PolyLib.dll
  
- FSharpApp F# console application
  - FSharpApp.fsproj includes reference to PolyLib
  - Program.fs a free-standing F# program

## Solution with three projects

- PolyLib

F# class library

- PolyLib.fsproj includes compilation information
- Polynomial.fsi signature (interface) file
- Polynomial.fs implementation file
- script.fsx reference to PolyLib.dll

- FSharpApp

F# console application

- FSharpApp.fsproj includes reference to PolyLib
- Program.fs a free-standing F# program

- CSharpApp

C# console application

- CSharpApp.csproj includes reference to PolyLib
- Program.cs a free-standing C# program  
Using the F# type `'a list`

# Overview of Solution

## Solution with three projects

- PolyLib F# class library
  - PolyLib.fsproj includes compilation information
  - Polynomial.fsi signature (interface) file
  - Polynomial.fs implementation file
  - script.fsx reference to PolyLib.dll
  
- FSharpApp F# console application
  - FSharpApp.fsproj includes reference to PolyLib
  - Program.fs a free-standing F# program
  
- CSharpApp C# console application
  - CSharpApp.csproj includes reference to PolyLib
  - Program.cs a free-standing C# program  
Using the F# type `'a list`

Let us have a look at the solution

## On Type Inference

Automated generation of the (most general) type for a program that do not contain type annotations.

Automated generation of the (most general) type for a program that do not contain type annotations.

- Avoid cluttering beautiful programs with type annotations while

Automated generation of the (most general) type for a program that do not contain type annotations.

- Avoid cluttering beautiful programs with type annotations while
- Preserving static type safety



# Type inference: Examples

For the two programs:

```
let f x y = 2 * x + y;;
```

```
let rec append xs ys = match xs with  
    | []          -> ys  
    | x::rest    -> x::append rest ys;;
```

the F# compiler infers the types:

```
f: int -> int -> int  
append: 'a list -> 'a list -> 'a list
```

# Type inference: Examples

For the two programs:

```
let f x y = 2 * x + y;;
```

```
let rec append xs ys = match xs with  
    | []          -> ys  
    | x::rest    -> x::append rest ys;;
```

the F# compiler infers the types:

```
f: int -> int -> int  
append: 'a list -> 'a list -> 'a list
```

The F# type inference includes

- **Overloading**

Functions with different types and implementations can share name, e.g. +

# Type inference: Examples

For the two programs:

```
let f x y = 2 * x + y;;
```

```
let rec append xs ys = match xs with  
    | []          -> ys  
    | x::rest    -> x::append rest ys;;
```

the F# compiler infers the types:

```
f: int -> int -> int  
append: 'a list -> 'a list -> 'a list
```

The F# type inference includes

- **Overloading**

Functions with different types and implementations can share name, e.g. `+`

- **Parametric polymorphism**

A single implementation of a function works for type-consistent input data, , e.g. `append`

# Type inference: Background

- Hindley type-schema for Combinatory Logic 1969
- Milner ML-style type inference with algorithm W 1978
- Damas-Milner 1982: correctness of type inference algorithm W
- ... SML ... OCAML ... Haskell ... F# ...

# Type inference: Background

- Hindley type-schema for Combinatory Logic 1969
- Milner ML-style type inference with algorithm W 1978
- Damas-Milner 1982: correctness of type inference algorithm W
- ... SML ... OCAML ... Haskell ... F# ...

Type inference in SML, ...F#, ... allows **let-polymorphism**; where, for example,

```
let rec map f xs = match xs with
  | []      -> []
  | x::xs -> f x :: map f xs
```

is typable  $\text{map} : \forall \alpha, \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$

# Type inference: Background

- Hindley type-schema for Combinatory Logic 1969
- Milner ML-style type inference with algorithm W 1978
- Damas-Milner 1982: correctness of type inference algorithm W
- ... SML ... OCAML ... Haskell ... F# ...

Type inference in SML, ...F#, ... allows **let-polymorphism**; where, for example,

```
let rec map f xs = match xs with
  | []      -> []
  | x::xs -> f x :: map f xs
```

is typable  $\text{map} : \forall \alpha, \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$

The following program is **NOT typable**:

```
let f g = (g 1, g true) in f id
```

as it would require two different instantiations of argument  $g$ 's type  
**such extra power makes type inference problem undecidable**

# About type rules and inference algorithm

The type-inference problem is specified using a few rules like:

$$\frac{\rho[x \mapsto t_x, f \mapsto t_x \rightarrow t_r] \vdash e_r : t_r \quad \rho[f \mapsto \forall \alpha_1, \dots, \alpha_n. t_x \rightarrow t_r] \vdash e_b : t}{\rho \vdash \text{let } f \ x = e_r \text{ in } e_b \text{ end} : t} \quad \text{rule 8 out of 9 (Sestoft2012)}$$

where  $\alpha_1, \dots, \alpha_n$  are not free in  $\rho$ .

don't worry

The identity function: `let id x = x` has infinity many types

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

including a **most general (or principal)** type `'a -> 'a`



The identity function: `let id x = x` has infinity many types

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

including a **most general (or principal)** type `'a -> 'a`

having all other types of `id` as instances

The identity function: `let id x = x` has infinity many types

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

including a **most general (or principal) type** `'a -> 'a`  
having all other types of `id` as instances

The step from a rule-based formulation to an algorithm is huge.

- rules and algorithm by Milner in 1978
- correctness proof of algorithm by Damas-Milner in 1982
- ML typability is complete for DEXPTIME by Mairson and KfouryTiurynUrzyczyn in 1990

The identity function: `let id x = x` has infinity many types

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

including a **most general (or principal) type** `'a -> 'a`  
having all other types of `id` as instances

The step from a rule-based formulation to an algorithm is huge.

- rules and algorithm by Milner in 1978
- correctness proof of algorithm by Damas-Milner in 1982
- ML typability is complete for DEXPTIME by Mairson and KfouryTiurynUrzyczyn in 1990

Nice presentations in

- Sestoft: Programming language concepts, Springer 2012 (Ch 6)
- Schwartzbach: Polymorphic type inference, BRICS LS 95-3

```
let pair x y p = p x y;;  
  
let p1 p = pair id id p;;  
let p2 p = pair p1 p1 p;;  
let p3 p = pair p2 p2 p;;  
let p4 p = pair p3 p3 p;;  
let p5 p = pair p4 p4 p;;
```

# A program with a complex type: Sestoft 2012

```
let pair x y p = p x y;;  
  
let p1 p = pair id id p;;  
let p2 p = pair p1 p1 p;;  
let p3 p = pair p2 p2 p;;  
let p4 p = pair p3 p3 p;;  
let p5 p = pair p4 p4 p;;
```

- `p1`'s type contains 3 type variables
- `p2`'s type contains 7 type variables
- `p3`'s type contains 15 type variables
- ...

# A program with a complex type: Sestoft 2012

```
let pair x y p = p x y;;  
  
let p1 p = pair id id p;;  
let p2 p = pair p1 p1 p;;  
let p3 p = pair p2 p2 p;;  
let p4 p = pair p3 p3 p;;  
let p5 p = pair p4 p4 p;;
```

- $p_1$ 's type contains 3 type variables
- $p_2$ 's type contains 7 type variables
- $p_3$ 's type contains 15 type variables
- ...

## Observe

- a doubling of the number of type variables from  $p_i$  to  $p_{i+1}$
- the number of type variable is exponential in the program size
- programmers rarely make programs having such complex types
- type checking appears to be efficient in practice

# An informal approach to type inference

Given a declaration

```
let rec f x y ... = e
```

and knowing typing and rules for “bits and pieces”.

# An informal approach to type inference

Given a declaration

```
let rec f x y ... = e
```

and knowing typing and rules for “bits and pieces”.

- Choose two fresh type variables for the unknown types of the arguments  $x, y, \dots$
- Analyse  $e$  adding new fresh type variables and constrains as needed when typing the parts of  $e$



# An informal approach to type inference

Given a declaration

```
let rec f x y ... = e
```

and knowing typing and rules for “bits and pieces”.

- Choose two fresh type variables for the unknown types of the arguments  $x, y, \dots$
- Analyse  $e$  adding new fresh type variables and constraints as needed when typing the parts of  $e$

Two possibilities:

- An inconsistency is detected and the program cannot be typed.
- A most general type can be establish as constraints on the introduced type variables arise from the program only.

## An example of an informal type inference

```
let rec map f xs = match xs with
  | []          -> []                (1)
  | x::tail    -> f x :: map f tail (2)
```

- Let  $'a$  and  $'b$  be fresh type variables so that  $f:'a$  and  $xs:'b$ .

## An example of an informal type inference

```
let rec map f xs = match xs with
  | []          -> []                (1)
  | x::tail    -> f x :: map f tail (2)
```

- Let  $'a$  and  $'b$  be fresh type variables so that  $f:'a$  and  $xs:'b$ .
- Since  $xs$  is matched with pattern  $[]$  in (1),  $xs$  must have type  $'c \text{ list}$  ( $'c$  fresh) and  $'b = 'c \text{ list}$

# An example of an informal type inference

```
let rec map f xs = match xs with
  | []          -> []                (1)
  | x::tail    -> f x :: map f tail (2)
```

- Let  $'a$  and  $'b$  be fresh type variables so that  $f:'a$  and  $xs:'b$ .
- Since  $xs$  is matched with pattern  $[]$  in (1),  $xs$  must have type  $'c \text{ list}$  ( $'c$  fresh) and  $'b = 'c \text{ list}$
- The value of the function must have type  $'d \text{ list}$  ( $'d$  fresh) due to the expression  $[]$  in (1)

# An example of an informal type inference

```
let rec map f xs = match xs with
  | []          -> []                (1)
  | x::tail    -> f x :: map f tail (2)
```

- Let  $'a$  and  $'b$  be fresh type variables so that  $f:'a$  and  $xs:'b$ .
- Since  $xs$  is matched with pattern  $[]$  in (1),  $xs$  must have type  $'c \text{ list}$  ( $'c$  fresh) and  $'b = 'c \text{ list}$
- The value of the function must have type  $'d \text{ list}$  ( $'d$  fresh) due to the expression  $[]$  in (1)
- Since  $xs:'c \text{ list}$  is matched with pattern  $x::tail$  in (2), we have  $x:'c$  and  $tail:'c \text{ list}$  due to the type of cons  $::$ .

# An example of an informal type inference

```
let rec map f xs = match xs with
  | []          -> []                (1)
  | x::tail    -> f x :: map f tail (2)
```

- Let  $'a$  and  $'b$  be fresh type variables so that  $f:'a$  and  $xs:'b$ .
- Since  $xs$  is matched with pattern  $[]$  in (1),  $xs$  must have type  $'c \text{ list}$  ( $'c$  fresh) and  $'b = 'c \text{ list}$
- The value of the function must have type  $'d \text{ list}$  ( $'d$  fresh) due to the expression  $[]$  in (1)
- Since  $xs:'c \text{ list}$  is matched with pattern  $x::tail$  in (2), we have  $x:'c$  and  $tail:'c \text{ list}$  due to the type of cons  $::$ .
- Since the value of the function has type  $'d \text{ list}$ , we have that  $f x::\text{map } f \text{ tail}:'d \text{ list}$  and hence  $f x:'d$ ,  $\text{map } f \text{ tail}:'d \text{ list}$  and  $f:'c \rightarrow 'd$  because  $x:'c$ .  
Therefore,  $'a = 'c \rightarrow 'd$ .

# An example of an informal type inference

```
let rec map f xs = match xs with
  | []          -> []                                (1)
  | x::tail    -> f x :: map f tail                  (2)
```

- Let  $'a$  and  $'b$  be fresh type variables so that  $f:'a \rightarrow 'd$  and  $xs:'b$ .
- Since  $xs$  is matched with pattern `[]` in (1),  $xs$  must have type  $'c \text{ list}$  ( $'c$  fresh) and  $'b = 'c \text{ list}$
- The value of the function must have type  $'d \text{ list}$  ( $'d$  fresh) due to the expression `[]` in (1)
- Since  $xs:'c \text{ list}$  is matched with pattern `x::tail` in (2), we have  $x:'c$  and  $tail:'c \text{ list}$  due to the type of `cons :: 'c -> 'c list -> 'c list`
- Since the value of the function has type  $'d \text{ list}$ , we have that  $f x::\text{map } f \text{ tail}:'d \text{ list}$  and hence  $f x:'d$ ,  $\text{map } f \text{ tail}:'d \text{ list}$  and  $f:'c \rightarrow 'd$  because  $x:'c$ .  
Therefore,  $'a = 'c \rightarrow 'd$ .

There are no further constraints and the most general type of `map` is

$('c \rightarrow 'd) \rightarrow 'c \text{ list} \rightarrow 'd \text{ list}$

Due to implicit universal quantification, the type can be renamed to

$('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

# An even more informal approach

Explain in your own words .....



# An even more informal approach

Explain in your own words .....

Explanation must

- justify that every sub-expression is well-typed and
- that the establish type is the most general one.

# An even more informal approach

Explain in your own words .....

Explanation must

- justify that every sub-expression is well-typed and
- that the establish type is the most general one.

An explanation concerning the type of `map` should address:

- the arguments `f` and `xs`,
- the patterns `[]` and `x::tail`,
- the expressions `[]` and `f x :: map f tail`,
- the sub-expressions `f x` and `map f tail`, and
- the type of `cons ::.`