02157 Functional Programming

Lecture 2: Functions, Types and Lists

Michael R. Hansen

DTU Compute

Department of Applied Mathematics and Computer Science

Outline

- Functional decomposition

short break

- Functions as ”first-class citizens”
  - Anonymous functions

short break

- Types, type inference and overloading
  - Tuples and equality and comparison constraints

short break

- Let expressions and lists

On functional decomposition

Functional decomposition

A simple technique when solving a complex problem is

- to partition it into smaller well-defined parts and, thereafter,
- to compose these parts to solve the original problem.

Functional decomposition

A simple technique when solving a complex problem is

- to partition it into smaller well-defined parts and, thereafter,
- to compose these parts to solve the original problem.

The main goal is that a program is constructed by

- combining simple well-understood pieces

Functional decomposition

A simple technique when solving a complex problem is

- to partition it into smaller well-defined parts and, thereafter,
- to compose these parts to solve the original problem.

The main goal is that a program is constructed by

- combining simple well-understood pieces

Invent useful helper functions

Functional decomposition: Example 1

Sorting a list:
`sort: 'a list -> 'a list when 'a : comparison`.

Functional decomposition: Example 1

Sorting a list:

`sort: 'a list -> 'a list when 'a : comparison`.

Insertion in an ordered list is easy:

```
(* insert:'a -> 'a list -> 'a list when 'a : comparison *)
```

Functional decomposition: Example 1

Sorting a list:
`sort: 'a list -> 'a list when 'a : comparison`.

Insertion in an ordered list is easy:

```
(* insert:'a -> 'a list -> 'a list when 'a : comparison *)
let rec insert x ys =
   match ys with
   | []                   -> [x]
   | y::_ when x <= y -> x::ys
   | y::ytail            -> y::insert x ytail
```

Functional decomposition: Example 1

Sorting a list:
`sort: 'a list -> 'a list when 'a : comparison`.

Insertion in an ordered list is easy:

```
(* insert:'a -> 'a list -> 'a list when 'a : comparison *)
let rec insert x ys =
   match ys with
   | []                   -> [x]
   | y::_ when x <= y -> x::ys
   | y::ytail             -> y::insert x ytail
```

Insertion sort can now easily be implemented:

```
let rec sort xs =
    match xs with
    | []          -> []
    | x::xtail -> insert x (sort xtail)
```

Functional decomposition: Example 1

Sorting a list:
`sort: 'a list -> 'a list when 'a : comparison`.

Insertion in an ordered list is easy:

```
(* insert:'a -> 'a list -> 'a list when 'a : comparison *)
let rec insert x ys =
   match ys with
   | []                -> [x]
   | y::_ when x <= y -> x::ys
   | y::ytail          -> y::insert x ytail
```

Insertion sort can now easily be implemented:

```
let rec sort xs =
    match xs with
    | []       -> []
    | x::xtail -> insert x (sort xtail)
```

- small comprehensible programs

Functional decomposition: Example 1

Sorting a list:

`sort: 'a list -> 'a list when 'a : comparison`.

Insertion in an ordered list is easy:

```
(* insert:'a -> 'a list -> 'a list when 'a : comparison *)
let rec insert x ys =
   match ys with
   | []                  -> [x]
   | y::_ when x <= y -> x::ys
   | y::ytail            -> y::insert x ytail
```

Insertion sort can now easily be implemented:

```
let rec sort xs =
    match xs with
    | []       -> []
    | x::xtail -> insert x (sort xtail)
```

- small comprehensible programs

Notice: there are better sorting algorithms than insertion sort

Functional decomposition: Example 2

Consider multiplication of polynomials:

$$
\begin{aligned}
0 \cdot Q(x) \quad &= 0 \\
(a_0 + a_1 \cdot x + ... + a_n \cdot x^n) \cdot Q(x) & \\
&= a_0 \cdot Q(x) + x \cdot \big((a_1 + a_2 \cdot x + ... + a_n \cdot x^{n-1}) \cdot Q(x)\big)
\end{aligned}
$$

Functional decomposition: Example 2

Consider multiplication of polynomials:

$$
\begin{aligned}
&0 \cdot Q(x) \quad = 0 \\
&(a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n) \cdot Q(x) \\
&\qquad = a_0 \cdot Q(x) + x \cdot \left((a_1 + a_2 \cdot x + \ldots + a_n \cdot x^{n-1}) \cdot Q(x)\right)
\end{aligned}
$$

Invent suitable auxiliary functions

- $a_0 \cdot Q(x)$ `mulC: int -> Poly -> Poly`

Functional decomposition: Example 2

Consider multiplication of polynomials:

$$
\begin{aligned}
0 \cdot Q(x) &= 0 \\
(a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n) \cdot Q(x) & \\
&= a_0 \cdot Q(x) + x \cdot \big((a_1 + a_2 \cdot x + \ldots + a_n \cdot x^{n-1}) \cdot Q(x)\big)
\end{aligned}
$$

Invent suitable auxiliary functions

- $a_0 \cdot Q(x)$ `mulC: int -> Poly -> Poly`
- $x \cdot (\ldots)$ `mulX: Poly -> Poly`

Functional decomposition: Example 2

Consider multiplication of polynomials:

$$
\begin{aligned}
&0 \cdot Q(x) \quad = 0 \\
&(a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n) \cdot Q(x) \\
&\qquad\qquad = a_0 \cdot Q(x) + x \cdot \big((a_1 + a_2 \cdot x + \ldots + a_n \cdot x^{n-1}) \cdot Q(x)\big)
\end{aligned}
$$

Invent suitable auxiliary functions

- $a_0 \cdot Q(x)$ `mulC: int -> Poly -> Poly`
- $x \cdot (\ldots)$ `mulX: Poly -> Poly`
- $a_0 \cdot Q(x) + x \cdot (\ldots)$ `add: Poly -> Poly -> Poly`

Functional decomposition: Example 2

Consider multiplication of polynomials:

$$
\begin{aligned}
&0 \cdot Q(x) \quad = 0 \\
&(a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n) \cdot Q(x) \\
&\qquad = a_0 \cdot Q(x) + x \cdot \big((a_1 + a_2 \cdot x + \ldots + a_n \cdot x^{n-1}) \cdot Q(x)\big)
\end{aligned}
$$

Invent suitable auxiliary functions

- $a_0 \cdot Q(x)$ `mulC: int -> Poly -> Poly`
- $x \cdot (\ldots)$ `mulX: Poly -> Poly`
- $a_0 \cdot Q(x) + x \cdot (\ldots)$ `add: Poly -> Poly -> Poly`

That makes the task of declaring multiplication easier

Invent helper function(s): Blackboard exercise

Declare a function `sumProd: int list -> int*int`:

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

$$\texttt{sumProd } [\,] = (0,1)$$

On functions as first-class citizens

Functions as ”first-class citizens”

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

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.

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

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.

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 parameterized programs
- Program libraries typically contain many such functions

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
```

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
```

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
```

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

Currying and Uncurrying

The process of turning a function on pairs (tuples) into a higher-order function is called currying. The opposite process is called uncurrying.

Currying and Uncurrying

The process of turning a function on pairs (tuples) into a higher-order function is called currying. The opposite process is called uncurrying.

Consider declarations:

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

let wUC(ro, s) = ro * s ** 3.0;;
val wUC : ro:float * s:float -> float
```

- `wC` is the curried version of `wUC`
- `wUC` is the uncurried version of `wC`

Currying and Uncurrying

The process of turning a function on pairs (tuples) into a higher-order function is called currying. The opposite process is called uncurrying.

Consider declarations:

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

let wUC(ro, s) = ro * s ** 3.0;;
val wUC : ro:float * s:float -> float
```

- `wC` is the curried version of `wUC`
- `wUC` is the uncurried version of `wC`

Have a look at exercise HR 2.13:

- declare functions for curring and uncurring.

```
curry:   ('a * 'b -> 'c) -> 'a -> 'b -> 'c
uncurry: ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

A well-known example: function composition

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

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

A well-known example: function composition

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
```

- An infix operator appears between the arguments

Infix functions

The prefix version ($(\oplus)$) of an infix operator $\oplus$ is a curried function, that is, higher-order function where argument are supplied one by one

For example:

```
(<<);;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
```

Infix functions

The prefix version $(\oplus)$ of an infix operator $\oplus$ is a curried function, that is, higher-order function where argument are supplied one by one

For example:

```
(<<);;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
```

- The argument is a function `'a -> 'b`

Infix functions

The prefix version ($(\oplus)$) of an infix operator $\oplus$ is a curried function, that is, higher-order function where argument are supplied one by one

For example:

```
(<<);;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
```

- The argument is a function `'a -> 'b`
- The value is a function `('c -> 'a) -> 'c -> 'b`

Infix functions

The prefix version $(\oplus)$ of an infix operator $\oplus$ is a curried function, that is, higher-order function where argument are supplied one by one

For example:

```
(<<);;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
```

- The argument is a function `'a -> 'b`
- The value is a function `('c -> 'a) -> 'c -> 'b`

Declaration

`<<` is a built-in function

```
let (<<) f g x = f(g x);;
```

Infix functions

The prefix version (⊕) of an infix operator ⊕ is a curried function, that is, higher-order function where argument are supplied one by one

For example:

```
(<<);;
val it : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
```

- The argument is a function `'a -> 'b`
- The value is a function `('c -> 'a) -> 'c -> 'b`

Declaration

`<<` is a built-in function

```
let (<<) f g x = f(g x);;
```

Infix operators are written as strings of special characters including

```
!  %  &  *  +  -  /  <  =  >  ?  @  ^  \  ~
```

Consult F# specification for complete rules.

The built-in infix function `@`

List.append is a higher-order function from the `List` library:

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

The built-in infix function `@`

List.append is a higher-order 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 built-in infix function `@`

List.append is a higher-order 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 ( @ ) : 'a list -> 'a list -> 'a list
```

The built-in infix function `@`

List.append is a higher-order 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 ( @ ) : 'a list -> 'a list -> 'a list
```

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

On anonymous functions

Function expressions

There are two kinds of expression for anonymous functions

- One originates from abstraction $\lambda x.e$ in the lambda calculus:

  `fun x` $\rightarrow$ ***e***

  reads: “the function of *x* given by *e*”.

Function expressions

There are two kinds of expression for anonymous functions

- One originates from abstraction $\lambda x.e$ in the lambda calculus:

  fun x $\rightarrow$ *e*

  reads: "the function of *x* given by *e*".

- The other support pattern matching:

  function
  | $pat_1 \rightarrow e_1$
  $\vdots$
  | $pat_n \rightarrow e_n$

You can write functions without naming them

Anonymous functions: Example 1

An expressions denoting the circle-area function

```
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: Example 2

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>
```

Anonymous functions: Example 2

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
```

Anonymous functions: Example 3

Function expressions with general patterns, e.g.

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

Anonymous functions: Example 3

Function expressions with general patterns, e.g.

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

Exploits an or pattern in the second clause

Simple functions expressions with *currying*

The expression

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

has the same meaning as

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

Simple functions expressions with *currying*

The expression

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

has the same meaning as

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

It denotes a function with type

$t_x \rightarrow (t_y \rightarrow (\cdots (t_z \rightarrow t_e) \cdots))$ where $x : t_x, y : t_y, z : t_z$ and $e : t_e$

Simple functions expressions with *currying*

The expression

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

has the same meaning as

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

It denotes a function with type

$$t_x \rightarrow (t_y \rightarrow (\cdots (t_z \rightarrow t_e) \cdots)) \text{ where } x : t_x, y : t_y, z : t_z \text{ and } e : t_e$$

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>
```

Simple functions expressions with *currying*

The expression

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

has the same meaning as

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

It denotes a function with type

$t_x \to (t_y \to (\cdots(t_z \to t_e)\cdots))$ where $x : t_x, y : t_y, z : t_z$ and $e : t_e$

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)
```

Simple functions expressions with *currying*

The expression

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

has the same meaning as

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

It denotes a function with type

$$t_x \rightarrow (t_y \rightarrow (\cdots(t_z \rightarrow t_e)\cdots)) \text{ where } x : t_x, y : t_y, z : t_z \text{ and } e : t_e$$

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
```

On types, type inference and overloading

Types and type checking

Purposes:

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

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

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

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
  - Checks of types are not needed at runtime

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
  - Checks of types are not needed at runtime

A type checker is an algorithm used at an early phase in the compiler to reject programs containing type errors.

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
  - Checks of types are not needed at runtime

A type checker is an algorithm used at an early phase in the compiler to reject programs containing type errors.

Type inference is an algorithm to automatically calculate types of expressions without use of explicit type annotations.

Fundamental type-checking problem

All non-trivial semantic properties of programs are undecidable

Rice's theorem

Fundamental type-checking problem

All non-trivial semantic properties of programs are undecidable

Rice's theorem

Example: *p* terminates on all its input

Fundamental type-checking problem

All non-trivial semantic properties of programs are undecidable

Rice's theorem

Example: *p* terminates on all its input

Cannot be checked for programs belonging to Turing-powerful languages

Fundamental type-checking problem

All non-trivial semantic properties of programs are undecidable

Rice's theorem

Example: *p* terminates on all its input

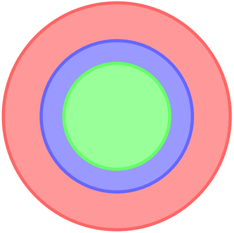
Cannot be checked for programs belonging to Turing-powerful languages

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, ...

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
```

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.

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.

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

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.

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

Remark: identity function id is a built-in function

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 *)
```

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 $ys : {'v}$

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

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 exploiting the type of `::` — C 2

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 of `::`
  3. ys : 'a list, 'v = 'a list — C 1,2
     ys must have the same type as x::(xtail @ ys)

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 of `::`
  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
```

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 of `::`
  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

Parametric and ad-hoc polymorphism - Strachey 1967

- Parametric polymorphism:
  – a function can be written so that it handles values identically independent on their types

  preserving type safety

Parametric and ad-hoc polymorphism - Strachey 1967

- Parametric polymorphism:
  – a function can be written so that it handles values identically independent on their types

preserving type safety

For example, the same code for append can be used for integer lists, list of pairs, list of ...

Parametric and ad-hoc polymorphism - Strachey 1967

- Parametric polymorphism:
  – a function can be written so that it handles values identically independent on their types

preserving type safety

For example, the same code for append can be used for integer lists, list of pairs, list of ...

- Ad-hoc polymorphism/overloading:
  – a function has different implementations depending on the type of arguments

For example, + can be used on integers, floating-points values, strings, ...

Overloaded Operators and Type inference

A squaring function on integers:

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

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	

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

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

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

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

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

On tuples and equality and comparison constraints

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:

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;2]; [3;4;5]] = [[1..2]; [3..5]];;
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;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
```

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)

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
```

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

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
```

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
```

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
```

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
```

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 a recursion following the structure of lists

On let-expressions and lists

Let-expressions

A let-expression e_l has the (verbose) form

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

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.

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:

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. v_1 is the value obtained by evaluating e_1 in *env*,

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. v_1 is the value obtained by evaluating e_1 in *env*,
2. env' is obtained by adding binding $x \mapsto v_1$ to *env* and

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. v_1 is the value obtained by evaluating e_1 in *env*,
2. env' is obtained by adding binding $x \mapsto v_1$ to *env* and
3. v_2 is the value obtained by evaluating e_2 in env'

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. v_1 is the value obtained by evaluating e_1 in env,
2. env' is obtained by adding binding $x \mapsto v_1$ to env and
3. v_2 is the value obtained by evaluating e_2 in env'

then

$$(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, env) \leadsto (v_2, env)$$

Let-expression – an example

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

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

Note: a and b are not visible outside of g

Let-expression – an example

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

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

Note: a and b are not visible outside of g

Evaluation ?

Pattern matching on results of recursive calls

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

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

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}`])`

Function expressions and match expressions

Consider

$$\text{Let } e_m \text{ be } \left\{ \begin{array}{lcl} \texttt{match}\ e\ \texttt{with} & & \\ \mid pat_1 & \to & e_1 \\ & \vdots & \\ \mid pat_n & \to & e_n \end{array} \right.$$

and

$$\text{Let } e_f \text{ be } \left\{ \begin{array}{lcl} \texttt{function} & & \\ \mid pat_1 & \to & e_1 \\ & \vdots & \\ \mid pat_n & \to & e_n \end{array} \right.$$

Function expressions and match expressions

Consider

$$\text{Let } e_m \text{ be } \left\{ \begin{array}{l} \texttt{match } e \texttt{ with} \\ \;\mid pat_1 \quad \rightarrow \quad e_1 \\ \qquad\qquad \vdots \\ \;\mid pat_n \quad \rightarrow \quad e_n \end{array} \right.$$

and

$$\text{Let } e_f \text{ be } \left\{ \begin{array}{l} \texttt{function} \\ \;\mid pat_1 \quad \rightarrow \quad e_1 \\ \qquad\qquad \vdots \\ \;\mid pat_n \quad \rightarrow \quad e_n \end{array} \right.$$

- Can you express e_f using e_m and ... ?
- Can you express e_m using e_f and ... ?

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x $\_$ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions: x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $[p_1; \ldots; p_n]$

  $e_1\, e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses* ...

where the construct *clauses* has the form:

$|\ p_1\ \texttt{->}\ e_1\ |\ \ldots\ |\ p_n\ \texttt{->}\ e_n$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x `_` $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions: x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $[p_1; \ldots; p_n]$

  $e_1\, e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses* ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$

where the construct *clauses* has the form:

`|` p_1 `->` e_1 `| ... |` p_n `->` e_n

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions: x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $[p_1; \ldots; p_n]$

  $e_1\, e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses*   ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int` `float` `bool` `string` $'a \ldots$
  $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2 \ldots$

where the construct *clauses* has the form:

`|` p_1 `->` e_1 `| ... |` p_n `->` e_n

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x `_` $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions: x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $[p_1; \ldots; p_n]$

  $e_1\, e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses* ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int` `float` `bool` `string` $'a \ldots$
  $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2 \ldots$

where the construct *clauses* has the form:

| p_1 -> e_1 | ... | p_n -> e_n

In addition to that

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

Summary

- Functional decomposition
- Functions as ”first-class citizens”
- Anonymous functions
- Types, type inference and overloading
- Tuples and equality and comparison constraints
- Let expressions and lists