Option Values, Arrays, Sequences, and Lazy Evaluation

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`
`http://www.idt.mdh.se/˜blr/`

The Option Data Type

A builtin data type in F#

Would be defined as follows:

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

A polymorphic type: for every type `t`, there is an option type `t option`

Option data types add an extra element `None`

Can be used to represent:

- the result of an erroneous computation (like division by zero)
- the absence of a “real” value

An Example: List.tryFind

`List.tryFind : ('a -> bool) -> 'a list -> 'a option`

A standard function in the `List` module

Takes a predicate p and a list `l` as arguments

Returns the first value in `l` for which p becomes true, or `None` if such a value doesn't exist in `l`

```
let rec tryFind p l =
  match l with
  | []              -> None
  | x::xs when p x -> Some x
  | _::xs           -> tryFind p xs
```

`List.tryFind even [1;3;8;2;5]` $\Longrightarrow$ `Some 8`

`List.tryFind even [1;3;13;13;5]` $\Longrightarrow$ `None`

`None` marks the failure of finding a value that satisfies the predicate. The caller can then take appropriate action if this situation occurs:

```
match List.tryFind p l with
| Some x -> x
| None   -> failwith "No element found!"
```

Another Example: a Zip Which Does Not Drop Values

`List.zip` requires that the lists are equally long

Let's define a version that works with lists of different length, and that does not throw away any elements. `None` represents the absence of a value. Elements at the end of the longer list are paired with `None`

It should work like this:

```
zippo [1;2;3] ['a';'b'] ⟹
[(Some 1, Some 'a');(Some 2, Some 'b');(Some 3, None)]
```

Solution on next slide . . .

Solution

```
zippo : ’a list -> ’b list -> (’a option * ’b option) list
let rec zippo l1 l2 =
  match (l1,l2) with
  | (a::as,b::bs) -> (Some a, Some b) :: zippo as bs
  | (a::as,[])) -> (Some a, None) :: zippo as []
  | ([],b::bs) -> (None, Some b) :: zippo [] bs
  | _                  -> []
```

Arrays

F# has arrays

For any F# type `'a`, there is a type `'a []`

F# arrays provide an alternative to lists

Sometimes arrays are better to use, sometimes lists are better

Some Properties of F# Arrays

Created with *fixed size*

Can be *multidimensional* (won't be brought up here)

Storage-efficient

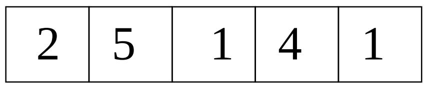
Constant lookup time

Mutable (elements can be updated, we'll bring this up later)

No sharing (different arrays are always stored separately)

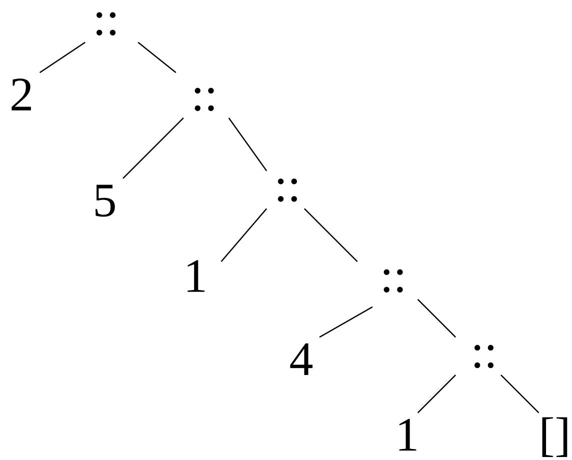
Arrays vs. Lists (1/2)

Arrays



Fixed size, small overhead, constant time random access, no sharing

Lists



Easy extension (cons), some memory overhead, access time grows with depth, sharing possible

Arrays vs. Lists (2/2)

Arrays are good when:

- the size is known in advance
- low access times to arbitrary elements are important
- low memory consumption is important
- no or little sharing is possible

Lists are good when:

- It is hard to predict the size in advance
- It is natural to build the data structure successively by adding elements
- there are opportunities for sharing

Creating and Accessing Arrays

Arrays can be created with a syntax very similar to list notation:

```
let a = [|1;2;1;5;0|]
a : int []
```

Creates an integer array of size 5

Accessing element `i`: `a.[i]`

`a.[0]` $\Longrightarrow$ 1 (arrays are indexed from `0`)

Accessing a *slice* (subarray): `a.[i..j]`

`a.[1..3]` $\Longrightarrow$ `[|2;1;5|]`

Empty array: `[||]`

Arrays vs. Strings

Elements and slices in arrays are accessed exactly as from strings

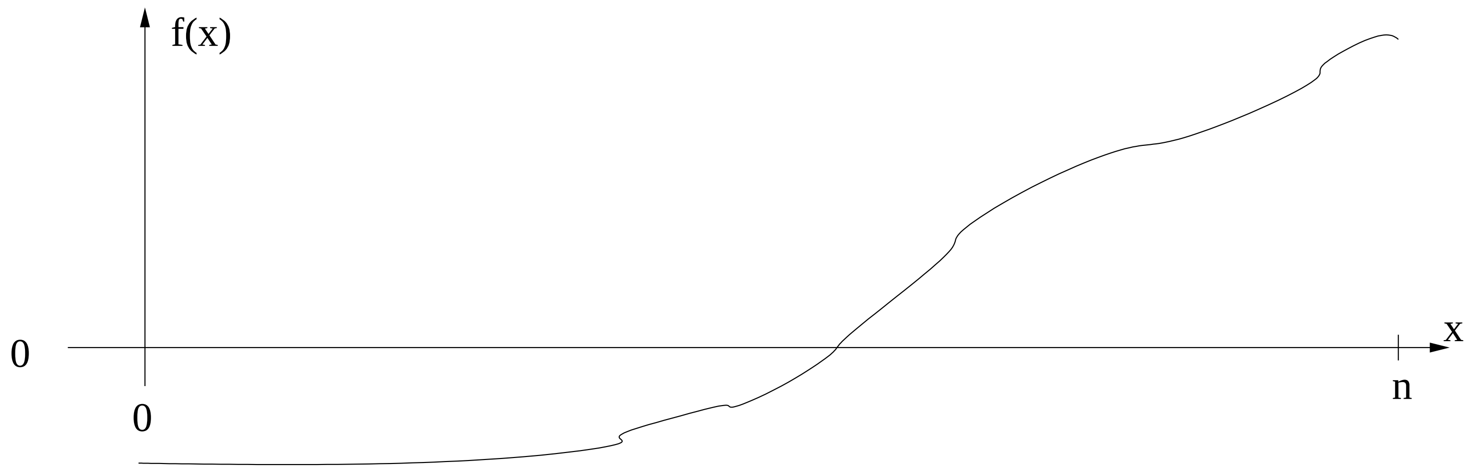
However, strings are *not* arrays of characters!

`string` $\neq$ `char []`

Also strings are immutable, whereas arrays of chars are mutable

An Array Programming Example

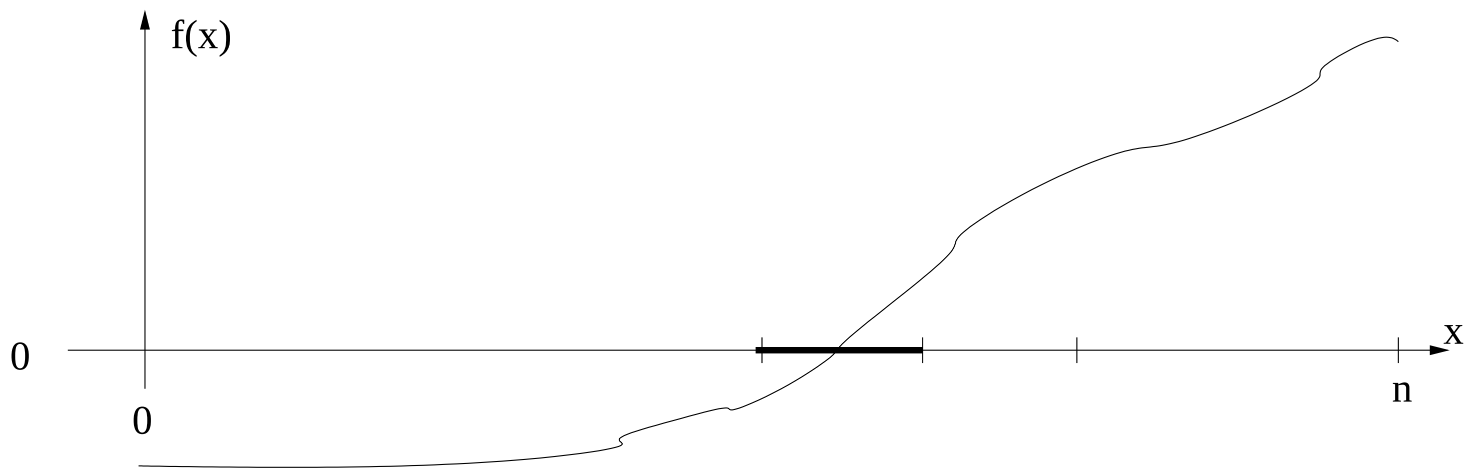
Problem: we have a mathematical (numerical) function f. We want to solve the equation $f(x) = 0$ numerically



We assume that: f is increasing on the interval $[0, n]$, that $f(0) \leq 0$, that $f(n) \geq 0$, and that f is continuous. Then $f(x) = 0$ has exactly one solution on the interval

A Classical Method: Interval Halving

By successively halving the interval, we can “close in” the value of x for which $f(x) = 0$



We stop when the interval is sufficiently narrow

Now assume that the function values $f(0), f(1), \ldots, f(n)$ are stored as a table, in an array a with $n + 1$ elements

We can then apply interval halving on the table. We define a recursive function that starts with `(0,n)` and recursively halves the interval. We stop when:

- we have an interval `(l,u)` where `a.[l] = 0.0`
- we have an interval `(l,u)` where `a.[u] = 0.0`
- we have an interval `(l,l+1)`

Solution (I)

Two possible results:

- An exact solution is found (`a.[i] = 0.0` for some `i`)
- The solution is enclosed in an interval `(l,l+1)`

Let's roll a data type to help distinguish these:

```
type Answer = Exact of int | Interval of int * int
```

Solution (II)

```
let rec int_halve (a : float []) l u =
  if u = l+1 then Interval (l,u)
  elif a.[l] = 0.0 then Exact l
  elif a.[u] = 0.0 then Exact u
  else let h = (l+u)/2 in
       if a.[h] > 0.0 then int_halve a l h
                       else int_halve a h u
```

Four cases to handle

Note the “`elif`” syntax, convenient for nested if:s

(For some reason we need to type a explicitly)

The Array Module

F# has an `Array` module, similar to the `List` module

Some standard array functions:

```
Array.length : ’a [] -> int
Array.append : ’a [] -> ’a [] -> ’a []
Array.zip : ’a [] -> ’b [] -> (’a * ’b) []
Array.filter : (’a -> bool) -> ’a [] -> ’a []
Array.map : (’a -> ’b) -> ’a [] -> ’b []
Array.fold : (’a -> ’b -> ’a) -> ’a -> ’b [] -> ’a
Array.foldBack : (’a -> ’b -> ’b) -> ’a [] -> ’b -> ’b
```

These work like their list counterparts. The above is just a selection. Notably no head, tail, or “cons” for arrays

An Observation on the Array Functions

Many of the array functions have exact counterparts for lists

This is not a coincidence

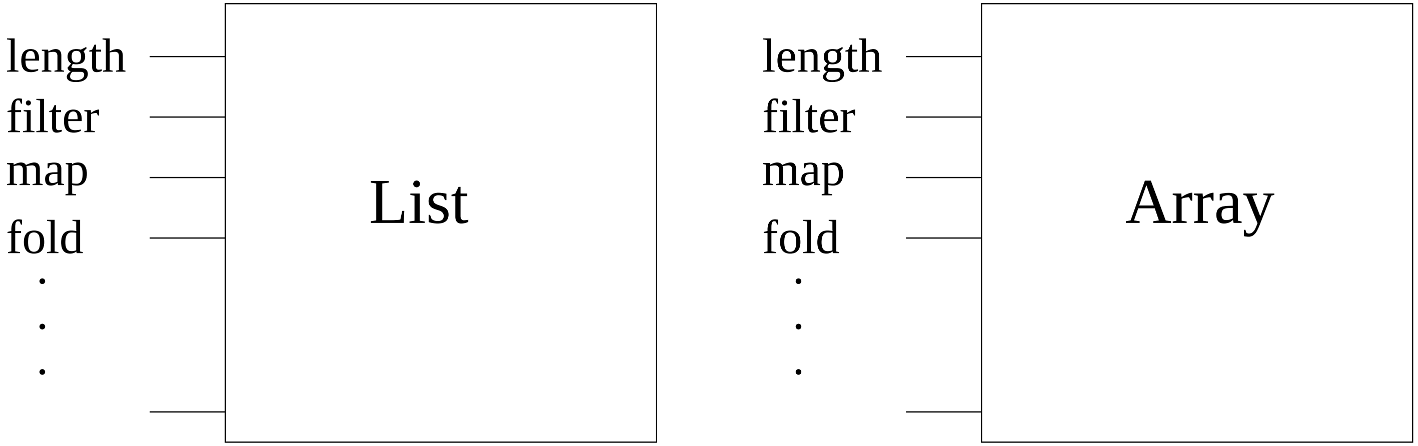
Arrays and lists just provide different ways to store *sequences of values*

Many of the functions, like `map`, `fold`, `filter`, etc. are really mathematical functions on sequences

So for *any* datatype that stores sequences, these functions can be defined

Software that uses these primitives can therefore easily be modified to use different data representations

Abstract Data Types



`length`, `map`, `fold` etc. provide an *interface*

It turns `List` and `Array` into *abstract data types*

If the programmer sticks to the interface, then *any* abstract data type implementing the interface can be used

An Example: Computing Mean Values

The mean value of n values $x_1, \ldots, x_n$ is defined as:

$$(\sum_{i=1}^{n} x_i)/n$$

A function to calculate the mean value of the elements in an array of floats:

```
let mean x = Array.fold (+) 0.0 x/float (Array.length x)
```

A little home exercise: change `mean` to calculate the mean value of a list of floats. Hint: it can be done quickly . . .

Sequences

F# has a data type for `seq<’a>` for *sequences* of values of type `’a`

Underneath, this is really the .NET type
`System.Collection.Generic.IEnumerable<’a>`

In F#, sequences are used:

- as an abstraction for lists and arrays,
- as a compute-on-demand construct, especially for interfacing with the outside world,

Sequences can be specified through *range* and *sequence expressions*

Range Expressions (1/2)

Range expressions are the simplest form of sequence expression:

```
{ start .. stop }
```

Generates a sequence with first element `start`, last element `stop`, and increment one

```
{ 1 .. 4 } ⟹ seq [1; 2; 3; 4] : seq<int>
{ 1.0 .. 4.0 } ⟹ seq [1.0; 2.0; 3.0; 4.0] : seq<float>
```

Primarily numerical types, but works for all types whose elements can be ordered:

```
{ ’a’ .. ’d’ } ⟹ seq [’a’; ’b’; ’c’; ’d’] : seq<char>
```

Range Expressions (2/2)

An increment can also be specified:

`{ start .. inc .. stop }`

`{ 1 .. 2 .. 8 }` $\Longrightarrow$ `seq [1; 3; 5; 7] : seq<int>`

Increments can be negative:

`{ 3.1 .. -0.5 .. 0.0 }` $\Longrightarrow$
`seq [3.1; 2.6; 2.1; 1.6; ...] : seq<float>`

(`fsi` only prints the first four elements of a sequence. Sequences are computed *on demand*, more on this later)

Some Functions on Sequences

F# has a module `Seq` with functions on sequences. Many of these have counterparts for lists and arrays. Some examples:

```
Seq.length : seq<’a> -> int
Seq.append : seq<’a> -> seq<’a> -> seq<’a>
Seq.take : int -> seq<’a> -> seq<’a>
Seq.skip : int -> seq<’a> -> seq<’a>
Seq.zip : seq<’a> -> seq<’b> -> seq<’a * ’b>
Seq.filter : (’a -> bool) -> seq<’a> -> seq<’a>
Seq.map : (’a -> ’b) -> seq<’a> -> seq<’b>
Seq.fold : (’a -> ’b -> ’a) -> ’a -> seq<’b> -> ’a
```

Examples:

```
Seq.map (fun i -> (i,i*i)) { 1 .. 100 } ⟹
seq [(1, 1); (2, 4); (3, 9); (4, 16); ...]
```

```
Seq.fold (+) 0 { 1 .. 100 } ⟹ 5050
```

Sequence Expressions (1/2)

A rich syntax for defining sequences

All of it is really syntactic sugar: can be done using the basic range expressions $+$ the functions in `Seq`. But convenient and easy to understand

A simple class of sequence expressions:

```
seq { for var in sequence -> expr }
```

Example:

```
seq { for i in 1 .. 100 -> (i,i*i) } ⟹
seq [(1, 1); (2, 4); (3, 9); (4, 16); ...]
```

(Same as `Seq.map (fun i -> (i,i*i)) { 1 .. 100 }`)

Sequence Expressions (2/2)

An extension:

```
seq { for pat in sequence -> expr }
```

Example:

```
let squares = seq { for i in 1 .. 100 -> (i,i*i) }
seq { for (i,i2) in squares -> (i2 - i) } ⟹
seq [0; 2; 6; 12; ...]
```

There are a number of other extensions

Lists, Arrays, and Sequences

`’a list` and `’a []` are *subtypes* to `seq<’a>`

This means that functions taking sequences as arguments can be given lists or arrays as arguments instead

Examples:

`Seq.map (fun x -> x+1) [1; 3; 5]` $\Longrightarrow$ `seq [2; 4; 6]`

`Seq.zip [|1; 3; 5|] [| ’a’ ;’b’ ;’c’|]` $\Longrightarrow$
`seq [(1, ’a’); (3, ’b’); (5, ’c’)]`

Defining Lists and Arrays by Sequence Expressions

Sequence expressions can be used to define lists or arrays

Simply write "`[ ... ]`" or "`[| ... |]`" rather than "`seq { ... }`"

$[1\ ..\ 5] \Longrightarrow [1;\ 2;\ 3;\ 4;\ 5]$
$[|1\ ..\ 5|] \Longrightarrow [|1;\ 2;\ 3;\ 4;\ 5|]$

Often very convenient for expressing predefined lists or arrays

Another example: converting an array to a list:

```
let array2list a = [for i in 0 .. Array.length a - 1 -> a.[i]]
```

Lazy Evaluation in F#

F# has two main constructs to yield lazy evaluation:

- a function `lazy` that delays evaluation, and a member `.Force()` that forces the evaluation
- sequences, which are computed on demand

Lazy/Force (1/4)

The `lazy` function in action (`fsi`):

```
> let x = lazy (33 + 12);;
val x : Lazy<int> = <unevaluated>
```

`x` obtains a special type `Lazy<int>`, and is unevaluated.

It is represented by a piece of code that will compute `33 + 12` when called

Lazy/Force (2/4)

`x` is evaluated with the `.Force()` member:

```
> x.Force();;
val it : int = 45
```

Subsequent evaluations of `x.Force()` will return the same value

Lazy/Force (3/4)

`x.Force()` evaluates `x` only *the first time it is called*

The value is stored, and reused: subsequent calls return the stored value

This becomes visible if we add a side effect:

```
> let x = lazy (printf "xxx\n"; 33 + 12);;
val x : Lazy<int> = <unevaluated>
> x.Force();;
xxx
val it : int = 45
> x.Force();;
val it : int = 45
```

The side effect occurs only the first time

Lazy/Force (4/4)

A comment on `lazy` and `.Force()`:

It is easy to do small examples with them

However, I found it hard to use them for more interesting things

I do think that the design of F# could be improved as regards laziness

Computation Expressions

F# has a concept of *computation expressions*

They can be used to fine-tune the order of evaluation

Sequence expressions are really computation expressions

We will not bring them up further here

See Ch. 12 in the book

Sequences

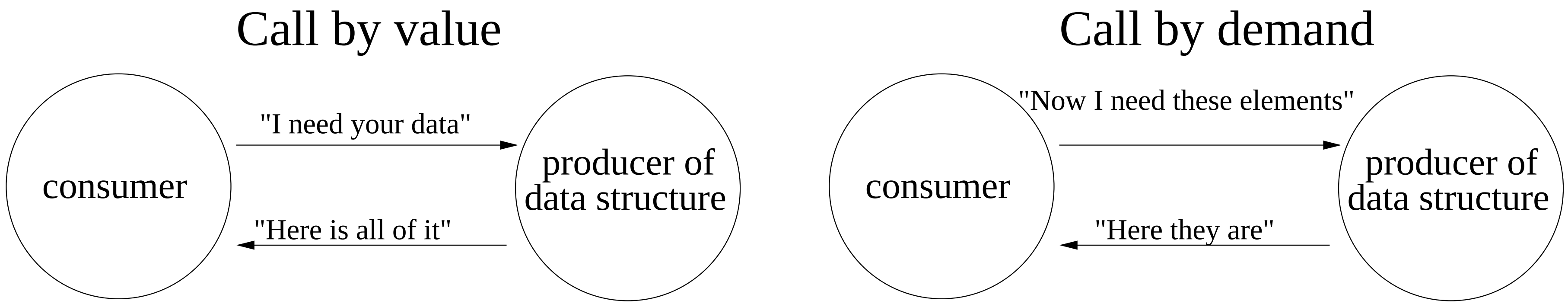
Sequences are computed *on demand*

Only as much as is “asked for” is computed

This means that we can work with very long, or even infinite sequences, as long as we only use a small part of them

Sequence functions like `Seq.take` and `Seq.tryFind` can be used to select small parts of sequences

Call by Value vs. Demand-driven Computation (1/2)



Call by Value vs. Demand-driven Computation (2/2)

Let's compare a list with 10 million elements with a sequence with 10 million elements

```
List.tryFind (fun x -> x = 3) [1 .. 10000000]
```

Call by value. The whole list will be evaluated, then searched for the first element that has the value `3`. The third element is returned

```
Seq.tryFind (fun x -> x = 3) { 1 .. 10000000 }
```

Evaluation by demand. `Seq.tryFind` will ask for elements one at a time, as it searches through the sequence. Only the three first elements will be generated, then `Seq.tryFind` returns the third element

Compare the performance in `fsi`!