Polymorphism, Recursive Data Types, Trees, and Option Values

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

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

Polymorphic types

Consider the good old `length` function:

```
let rec length l = match l with
                   | []      -> 0
                   | (x::xs) -> 1 + length xs
```

What is the type of `length`?

It could be `int list -> int`, or `char list -> int`, or even `(int list) list -> int`! So it has *many different types*!

`length` should really work regardless of the type of the elements

It has type `'a list -> int`, where `'a` is a *type variable*

`'a list -> int` is a *polymorphic type*

`length : 'a list -> int` means that `length` has any type we can obtain by replacing `'a` with some arbitrary type

Examples:

`'a` $\leftarrow$ `int` $\Longrightarrow$ `length : int list -> int`

`'a` $\leftarrow$ `char` $\Longrightarrow$ `length : char list -> int`

`'a` $\leftarrow$ `int list` $\Longrightarrow$ `length : (int list) list -> int`

`'a list -> int` is the *most general* type of `length`

The type system of F# gives the most general type, unless you give an explicit type declaration

Type inference is used to find this type

Some other polymorphic list functions (and lists):

```
List.head : 'a list -> 'a
List.tail : 'a list -> 'a list
take : int -> 'a list -> 'a list
drop : int -> 'a list -> 'a list
(@) : 'a list -> 'a list -> 'a list
(::)  : 'a -> 'a list -> 'a list
[]   : 'a list
```

A Restriction for Polymorphic Types

Some polymorphic expressions are not allowed

Due to some deep technical reasons

This is called the “value restriction”

Affects expressions that are not *value expressions*

A value expression can be evaluated no further. Some examples:

```
17  []  (2.3,[])  sqrt  [1;2;3] failwith
```

Some expressions that are not value expressions (can be evaluated further):

```
17+33  [] @ []  sqrt 5.0  List.head [1;2;3]  failwith "Error!"
```

The Value Restriction

The value restriction states that *right-hand sides in let declarations that are not value expressions can not be polymorphic*

Some examples:

```
let a = 17 + x
\\ OK, 17 + x is not a value expression but has type int

let b = []
\\ OK, [] has polymorphic type 'a list but is a value expression

let c = [] @ []
\\ Not OK, [] @ [] has polymorphic type and is not a value expression

let d = 3 :: ([] @ [])
\\ OK, 3 :: ([] @ []) has (non-polymorphic) type int list
```

Fixing the Value Restriction

The value restriction can often be overcome by an explicit type annotation to remove polymorphism:

```
let c = [] @ [] : int list
\\ OK, [] @ [] does not have a polymorphic type anymore
```

Sometimes some subexpressions can be evaluated to turn the right-hand side into a value expression

Example: evaluating `[] @ []` $\rightarrow$ `[]` in the declaration of `c` yields:

```
let c = []
\\ OK, [] is a value expression
```

Recursive Data Types

So far, we have defined data types with a number of cases, each of fixed size

How do we define data types for data like lists, which can have an arbitrary number of elements?

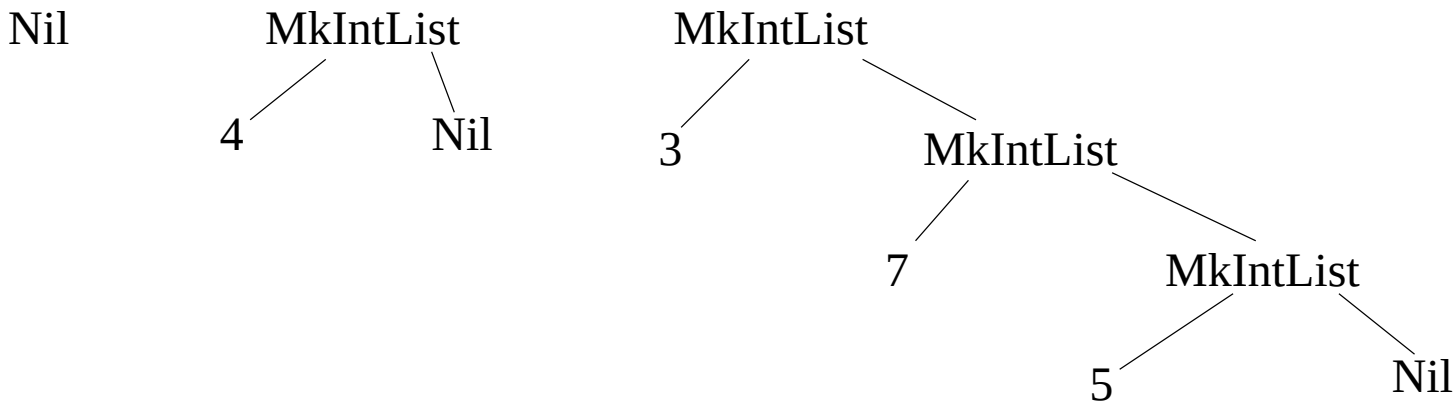
By making the data type definition *recursive*:

```
type IntList = Nil | MkIntList of (int * IntList)
```

An element of type `IntList` can be either `Nil`, or a data structure that contains an `int` and an `IntList`

Note similarity between data type declaration and context-free grammar

Some `IntList` examples:



Polymorphism

F#'s own data type for list is polymorphic

We can roll our own polymorphic list data type:

```
type List<'a> = Nil | MkList of 'a * List<'a>
```

Here, `'a` is a type variable. Note the syntax `<...>` for user-defined polymorphic types: different from syntax for built-in polymorphic data types like `'a list`

This data type is precisely the same as F#'s list data type, except that the constructor names are different!

Data type declarations can be recursive and polymorphic

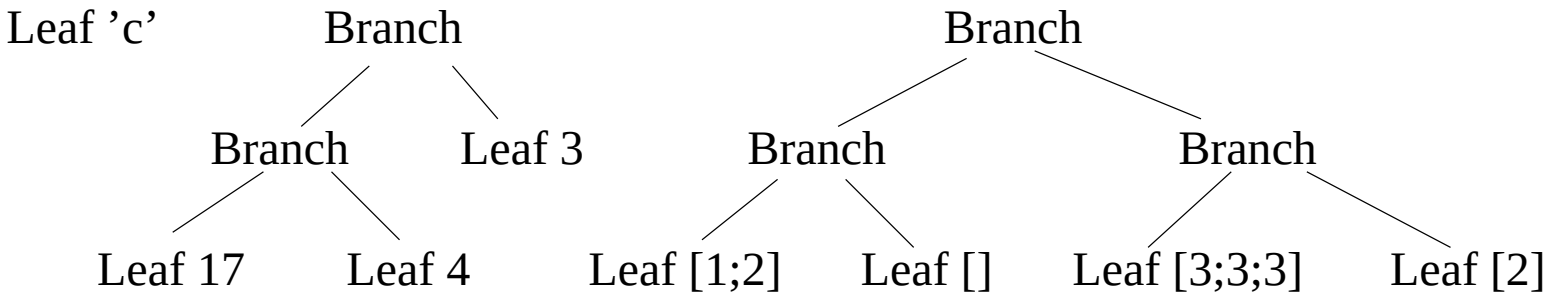
Most of F#'s built-in data types can in principle be declared in the language itself

Data Types for Trees

We can easily make our own data types for *trees*, like:

```
type Tree<'a> = Leaf of 'a | Branch of Tree<'a> * Tree<'a>
```

A data type for trees with data stored in the leaves



Many other variations possible, see examples in the book

Let us use this type for now

Operations on Trees

Let us define some useful operations over our trees:

- a function to put the elements in a tree into a list,
- a function to compute the *size* (number of leaves) of a tree, and
- a function to compute the *height* of a tree.

(Code on next two slides)

To put the elements in a tree into a list:

```
fringe : Tree<'a> -> 'a list
let rec fringe t =
  match t with
  | Leaf x          -> [x]
  | Branch (t1,t2) -> fringe t1 @ fringe t2
```

Size (number of leaves):

```
treeSize : Tree<'a> -> int
let rec treeSize t =
  match t with
  | Leaf _          -> 1
  | Branch (t1,t2) -> treeSize t1 + treeSize t2
```

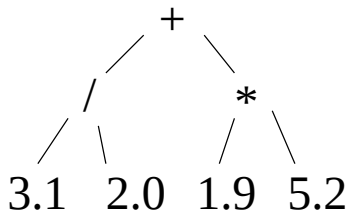
Height:

```
treeHeight : Tree<'a> -> int
let rec treeHeight t =
  match t with
  | Leaf _          -> 0
  | Branch (t1,t2) -> 1 + max (treeHeight t1) (treeHeight t2)
```

A Different Example: Arithmetic Expressions

Arithmetic expressions are really trees:

(3.1/2.0) + (1.9*5.2)



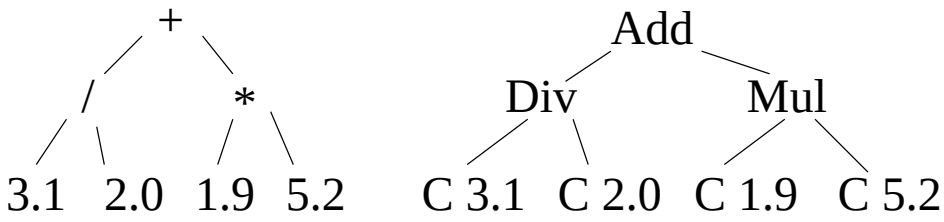
Let us define a data type for arithmetic (floating-point) expressions! We can then use it for various symbolic manipulations of such expressions

(Data type declaration on next slide)

```
type Expr = C of float | Add of Expr * Expr | Sub of Expr * Expr
          | Mul of Expr * Expr | Div of Expr * Expr
```

Each tree now represents an arithmetic expression:

(3.1/2.0) + (1.9*5.2)



Evaluating Expressions

One operation is to *evaluate* expressions

```
eval : Expr -> float
let rec eval e =
  match e with
  | C x -> x
  | Add (e1,e2) -> eval e1 + eval e2
  | Sub (e1,e2) -> eval e1 - eval e2
  | Mul (e1,e2) -> eval e1 * eval e2
  | Div (e1,e2) -> eval e1 / eval e2
```

`eval (Add ((C 17.0), Sub (C 3.0, C 1.0)))` $\Longrightarrow$ `19.0`

`eval` is a simple *interpreter* for our expression trees

Exercise (mini-project): extend `Expr` with *variables*. Then define a small *symbolic algebra package* for manipulating and simplifying expressions, for instance:

- evaluate constant subexpressions
- simplify as far as possible using algebraic identities
- symbolic derivation
- etc. . .

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   -> (appropriate action when no matching element was found)
```

Error Handling with Error Values

Reconsider the example from last slide:

```
match List.tryFind p l with
| Some x -> x
| None   -> (appropriate action when no matching element was found)
```

This shows how to use `None` as an *error value*

`failwith` will just break the computation, that is: a crash!

Error values can be examined and passed around. This allows for much smoother error handling

You can also define your own data types with error values:

```
type T = Error1 | Error2 | Error3 int | ...
```