ME 2017-10-14

14 | Programming with types

F# is a strongly typed language, meaning that types are known or inferred at compile time. In the previous chapters, we have used *primitive types* such as `float` and `bool`, function types, and compound types implicitly defined by tuples. These types are useful for simple programming tasks, and everything that can be programmed can be accomplished using these types. However, larger programs are often easier to read and write when using more complicated type structures. In this chapter, we will discuss type abbreviations, enumerated types, discriminated unions, records and structs. Class types are discussed in depth in Chapter 20.

· primitive types
maybe add lists

14.1 Type abbreviations

F# allows for renaming of types, which is called *type abbreviation* or *type aliasing*. The syntax is

· type abbreviatic
· type aliasing

Listing 14.1 Syntax for type abbreviation.

```
type <ident> = <aType>
```

where the identifier is a new name, and the type-name is an existing or a compound of existing types. E.g., in Listing 14.2 several type abbreviations are defined.

Listing 14.2 typeAbbreviation.fsx:
Defining 3 type abbreviations, two of which are compound types.

```
type size = int
type position = float * float
type person = string * int
type intToFloat = int -> float

let sz : size = 3
let pos : position = (2.5, -3.2)
let pers : person = ("Jon", 50)
let conv : intToFloat = fun a -> float a
printfn "%A, %A, %A, %A" sz pos pers (conv 2)
```

```
$ fsharpc --nologo typeAbbreviation.fsx && mono typeAbbreviation.exe
3, (2.5, -3.2), ("Jon", 50), 2.0
```

Here we define the abbreviations `size`, `position`, `person`, and `intToFloat`, and later make bindings

enforcing the usage of these abbreviations.

Type abbreviations are useful as short abbreviations of longer types, and they add semantic content to the program text, thus making programs shorter, easier to read. Type abbreviations allow the programmer to focus on the intended structure at a higher level by, e.g., programming in terms of a type `position` rather than `float * float`. Thus, they often results in programs with less errors. Type abbreviations also make maintenance easier, e.g., if we at a later stage decide that positions only can have integer values, then we only need to change the definition of the type abbreviation, not every place, a value of `position` type is used.

14.2 Enumerations

Enumerations or *enums* for short are types with named values. Names in enums are assigned to a subset of integer or char values. Their syntax is as follows:

- enumerations
- enums

Listing 14.3 Syntax for enumerations.

```
type <ident> =
  [ | ] <ident> = <integerOrChar>
  | <ident> = <integerOrChar>
  | <ident> = <integerOrChar>
  ...
```

An example of using enumerations is given in Listing 14.4.

Listing 14.4 enum.fsx:
An enum type acts as a typed alias to a set of integers or chars.

```
type medal =
  Gold = 0
  | Silver = 1
  | Bronze = 2

let aMedal = medal.Gold
printfn "%A has value %d" aMedal (int aMedal)
```

```
$ fsharpc --nologo enum.fsx && mono enum.exe
Gold has value 0
```

In the example, we define an enumerated type for medals, which allows us to work with the names rather than the values. Since the values most often are arbitrary, we can program using semantically meaningful names instead. Being able to refer to an underlying integer type allows us to interface with other – typically low-level – programs that requires integers, and to perform arithmetic. E.g., for the medal example, we can typecast the enumerated types to integers and calculate an average medal harvest.

14.3 Discriminated Unions

A discriminated union is a union of a set of named cases. These cases can further be of specified types. The syntax for defining a discriminated union is as follows:

Listing 14.5 Syntax for type abbreviation.

```
[<attributes>]
type <ident> =
  [| ]<ident> [of [<ident> :] <aType> [* [<ident> :] <aType> ...]]
  | <ident> [of [<ident> :] <aType> [* [<ident> :] <aType> ...]]
  ...
```

Discriminated unions are reference types, i.e., their content is stored on *The Heap*, see Section 6.7 for a discussion on reference types. Since they are immutable, ~~then~~ there is no risk of side-effects. As reference types, when used as arguments to and returned from a function, then only a reference is passed. This is in contrast to value types, which transport a complete copy of the data structure. Discriminated unions are thus effective for large data structures. However, there is a slight overhead, since working with the content of reference types is indirect through their reference. Discriminated unions can also be represented as structures using the `[<Struct>]` attribute, in which case they are value types. See Section 14.5 for a discussion on structs.

· The Heap

An example just using the named cases but no further specification of types is given in Listing 14.6.

Listing 14.6 discriminatedUnions.fsx:
A discriminated union of medals. Compare with Listing 14.4.

```
type medal =
  Gold
  | Silver
  | Bronze

let aMedal = medal.Gold
printfn "%A" aMedal
```

```
$ fsharpc --nologo discriminatedUnions.fsx
$ mono discriminatedUnions.exe
Gold
```

Here we define a discriminated union as three named cases signifying three different types of medals. Comparing with the enumerated type in Listing 14.4, we see that the only difference is that the cases of the discriminated unions have no value. A commonly used discriminated union is the *option type*, see Section 18.2 for more detail.

· option type

Discriminated unions may also be used to store data. Where the names in enumerated types are aliases of single values, the names in discriminated unions can hold any value specified at the time of creation. An example is given in Listing 14.7.

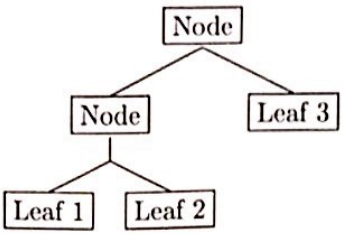

Figure 14.1: The tree with 3 leaves.

Listing 14.7 discriminatedUnionsOf.fsx:
A discriminated union using explicit subtypes.

```
type vector =
  Vec2D of float * float
  | Vec3D of x : float * y : float * z : float

let v2 = Vec2D (1.0, -1.2)
let v3 = Vec3D (x = 1.0, z = -1.2, y = 0.9)
printfn "%A and %A" v2 v3
```

```
$ fsharpc --nologo discriminatedUnionsOf.fsx
$ mono discriminatedUnionsOf.exe
Vec2D (1.0,-1.2) and Vec3D (1.0,0.9,-1.2)
```

In this case, we define a discriminated union of two and three-dimensional vectors. Values of these types are created using their names followed by a tuple of their arguments. As can be seen, the arguments may be given field names, and if they are, then the names may be used when creating values of this type. As also demonstrated, the field names can be used to specify the field values in arbitrary order. However, values for all fields must be given.

Discriminated unions can be defined recursively. This is shown in Listing 14.8.

Listing 14.8 discriminatedUnionTree.fsx:
A discriminated union modelling binary trees.

```
type Tree =
  Leaf of int
  | Node of Tree * Tree

let one = Leaf 1
let two = Leaf 2
let three = Leaf 3
let tree = Node (Node (one, two), three)
printfn "%A" tree
```

```
$ fsharpc --nologo discriminatedUnionTree.fsx
$ mono discriminatedUnionTree.exe
Node (Node (Leaf 1,Leaf 2),Leaf 3)
```

In this example we define a tree as depicted in Figure 14.1.

Pattern matching must be used in order to define functions on values of discriminated union. E.g., in

Listing 14.9 we define a function that traverses a tree and prints the content of the nodes.

Listing 14.9 discriminatedUnionPatternMatching.fsx: A discriminated union modelling binary trees.

```
type Tree = Leaf of int | Node of Tree * Tree
let rec traverse (t : Tree) : string =
    match t with
      Leaf(v) -> string v
      | Node(left, right) -> (traverse left) + ", " + (traverse right)

let tree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
printfn "%A: %s" tree (traverse tree)
```

```
$ fsharpc --nologo discriminatedUnionPatternMatching.fsx
$ mono discriminatedUnionPatternMatching.exe
Node (Node (Leaf 1,Leaf 2),Leaf 3): 1, 2, 3
```

But pattern-matching is not introduced yet!!!

Discriminated unions are very powerful and can often be used instead of class hierarchies. Class hierarchies are discussed in Section 20.10.

14.4 Records

A record is a compound of named values, and a record type is defined as follows:

Listing 14.10 Syntax for defining record types.

```
[ <attributes> ]
type <ident> = {
  [ mutable ] <label1> : <type1>
  [ mutable ] <label2> : <type2>
  ...
}
```

Records are like tuples in that they allow for representing a compound of values, but unlike tuples, for their content has names, and they are reference types, i.e., their content is stored on *The Heap*, see Section 6.7 for a discussion on reference types. Records can also be *struct records* using the [<Struct>] attribute, in which case they are value types. See Section 14.5 for a discussion on structs. An example of using records is given in Listing 14.11. The values of individual members of a record is obtained using the "." notation

· The Heap
· struct records
· ·

Is it really important here to distinguish between reference and value types? We don't need to provide the students with ALL features of the language; also, we don't need to introduce the [<attribute>] features at this stage!

Listing 14.11 records.fsx:
A record is defined for holding information about a person.

```
type person = {
    name : string
    age : int
    height : float
}

let author = {name = "Jon"; age = 50; height = 1.75}
printfn "%A\nname = %s" author author.name
```

```
$ fsharpc --nologo records.fsx && mono records.exe
{name = "Jon";
 age = 50;
 height = 1.75;}
name = Jon
```

Here is created a record of varied data about a person, and a name is bound to a record expression defining its field values.

If two record types are defined with the same label set, then the later dominates the former, and the compiler will at a binding infer that later. This is demonstrated in Listing 14.12.

Listing 14.12 recordsDominance.fsx:
Redefined types dominates old record types, but earlier definitions are still accessible using explicit or implicit specification for bindings.

```
type person = { name : string; age : int; height : float }
type teacher = { name : string; age : int; height : float }

let lecturer = {name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" lecturer (lecturer.GetType())
let author : person = {name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" author (author.GetType())
let father = {person.name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" author (author.GetType())
```

```
$ fsharpc --nologo recordsDominance.fsx && mono recordsDominance.exe
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+teacher
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+person
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+person
```

In the example, two identical record types are defined, and we use the builtin `GetType()` method to inspect the type of bindings. We see that `lecturer` is of `RecordsDominance+teacher` type, since `teacher` dominates the identical `author` type definition. However, we may enforce the `person` type by either specifying it for the name as in `let author : person = ...` or by fully or partially

specifying it in the record expression following the "=" sign. In both cases, they are therefore of `RecordsDominance+author` type. The built-in `GetType()` method is inherited from the base class for all types, see Chapter 20 for a discussion on classes and inheritance.

Note that when creating a record, you must supply a value to all fields, and you cannot refer to other fields of the same record, e.g., `{name = "Jon"; age = height * 3; height = 1.75}` is illegal.

Since records are per default reference types, binding creates aliases not copies. This matters for mutable members, in which case when copying, we must explicitly create a new record with the old data. Copying can be done either by using referencing to the individual members of the source or using the short-hand *with* notation. This is demonstrated in Listing 14.13. ·with

Listing 14.13 recordCopy.fsx:
Bindings are references. To copy and not make an alias, explicit copying must be performed.

```
1  type person = {
2    name : string;
3    mutable age : int;
4  }
5
6  let author = {name = "Jon"; age = 50}
7  let authorAlias = author
8  let authorCopy = {name = author.name; age = author.age}
9  let authorCopyAlt = {author with name = "Noj"}
10 author.age <- 51
11 printfn "author : %A" author
12 printfn "authorAlias : %A" authorAlias
13 printfn "authorCopy : %A" authorCopy
14 printfn "authorCopyAlt : %A" authorCopyAlt
```

```
1  $ fsharpc --nologo recordCopy.fsx && mono recordCopy.exe
2  author : {name = "Jon";
3   age = 51;}
4  authorAlias : {name = "Jon";
5   age = 51;}
6  authorCopy : {name = "Jon";
7   age = 50;}
8  authorCopyAlt : {name = "Noj";
9   age = 50;}
```

Here age is defined as a mutable value, and can be changed using the usual "<-" assignment operator. The example demonstrates two different ways to create records. Note that when the mutable value `author.age` is changed in line 10, then `authorAlias` also changes, since it is an alias of `author`, but neither `authorCopy` nor `authorCopyAlt` changes, since they are copies. As illustrated, copying using `with` allows for easy copying and partial updates of another record value.

14.5 Structures

Structures or *structs* for short have much in common with records. They specify a compound type with named fields, but they are value types, and they allow for some customization of what is to happen when a value of its type is created. Since they are value types, then they are best used for small amount of data. The syntax for defining struct types are: ·structures ·structs

Listing 14.14 Syntax for type abbreviation.

```
[ <attributes> ]
[<Struct>]
type <ident> =
  val [ mutable ] <label1> : <type1>
  val [ mutable ] <label2> : <type2>
  ...
  [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1> = <arg2>; ...}
  [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1> = <arg2>; ...}
  ...
```

The syntax makes use of the *val* and new keywords. Keyword val like let binds a name to a value, but unlike let the value is always the type's default value. The new keyword denotes the function used to fill values into the fields at time of creation. This function is called the *constructor*. No let nor do bindings are allowed in structure definitions. Fields are accessed using the "." notation. An example is given in Listing 14.15.

- val
- .
- new
- constructor
- .

Listing 14.15 struct.fsx:
Defining a struct type and creating a value of it.

```
[<Struct>]
type position =
  val x : float
  val y : float
  new (a : float, b : float) = {x = a; y = b}

let p = position (3.0, 4.2)
printfn "%A: x = %A, y = %A" p p.x p.y
```

```
$ fsharpc --nologo struct.fsx && mono struct.exe
Struct+position: x = 3.0, y = 4.2
```

Structs are small versions of classes and allows, e.g., for overloading of the new constructor and for overriding ~~override~~ of the inherited *ToString()* function. This is demonstrated in Listing 14.16.

- overload
- override
- ToString()

Listing 14.16 structOverloadNOverride.fsx: Overloading the constructor and overriding the default ToString() function.

```
[<Struct>]
type position =
  val x : float
  val y : float
  new (a : float, b : float) = {x = a; y = b}
  new (a : int, b : int) = {x = float a; y = float b}
  override this.ToString() =
    "(" + (string this.x) + ", " + (string this.y) + ")"

let pFloat = position (3.0, 4.2)
let pInt = position (3, 4)
printfn "%A and %A" pFloat pInt
```

```
$ fsharpc --nologo structOverloadNOverride.fsx
$ mono structOverloadNOverride.exe
(3, 4.2) and (3, 4)
```

We defer further discussion of these concepts to Chapter 20.

The use of structs are generally discouraged, and instead it is recommended to use enums, records, and discriminated unions possibly with the `[<Struct>]` attribute for the last two in order to make them value types.

14.6 Variable types

An advanced topic in F# is *variable types*. There are three different versions of variable types in F#: *runtime resolved*, which has the syntax `'<ident>`, *anonymous*, which are written as "`_`", and *statically resolved*, which have the syntax `^<ident>`. Variable types are particularly useful for functions that work for many types. An example of a generic function and its use is given in Listing 14.17.

- variable types
- runtime resolved variable type
- anonymous variable type
- _
- statically resolved variable type

Listing 14.17 variableType.fsx: A function apply with runtime resolved types.

```
let apply (f : 'a -> 'a -> 'a) (x : 'a) (y : 'a) : 'a = f x y
let intPlus (x : int) (y : int) : int = x + y
let floatPlus (x : float) (y : float) : float = x + y

printfn "%A %A" (apply intPlus 1 2) (apply floatPlus 1.0 2.0)
```

```
$ fsharpc --nologo variableType.fsx && mono variableType.exe
3 3.0
```

In this example, the function `apply` has runtime resolved variable type, and it accepts three parameters `f`, `x`, and `y`. The function will work as long as the parameters for `f` is a function of two parameters of identical type, and `x` and `y` are values of the same type. Thus, in the `printfn` statement, we are able to use `apply` for both an integer and a float variant.

The example in Listing 14.17 illustrates a very complicated way to add two numbers. And the "+" operator works for both types out of the box, so why not something simpler like relying on F# type inference system by not explicitly specifying types as attempted in Listing 14.18.

Listing 14.18 variableTypeError.fsx:
Even though the "+" operator is defined for both integers and floats, the type inference is static and infers `plus : int -> int`.

```
let plus x y = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo variableTypeError.fsx && mono variableTypeError.exe

variableTypeError.fsx(3,34): error FS0001: This expression was expected
   to have type
     'int'
but here has type
     'float'

variableTypeError.fsx(3,38): error FS0001: This expression was expected
   to have type
     'int'
but here has type
     'float'
```

Unfortunately, the example fails to compile, since the type inference is performed at compile time, and by `plus 1 2`, it is inferred that `plus : int -> int`. Hence, calling `plus 1.0 2.0` is a type error. Function bindings allow for the use of the *inline* keyword, and adding this successfully reuses the definition of `plus` for both types as shown in Listing 14.19.

Listing 14.19 variableTypeInline.fsx:
The keyword forces static and independent inference each place the function is used. Compare to the error case in Listing 14.18.

```
let inline plus x y = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo variableTypeInline.fsx && mono variableTypeInline.exe
3 3.0
```

In the example, adding the `inline` does two things: Firstly, it copies the code to be performed to each place, the function is used, and secondly, it forces statically resolved variable type checking independently in each place. The type annotations inferred as a result of the `inline`-keyword may be written explicitly as shown in Listing 14.20.

Listing 14.20 compiletimeVariableType.fsx:
Explicitely spelling out of the statically resolved type variables from Listing 14.18.

```
let inline plus (x : ^a) (y : ^a) : ^a when ^a : (static member ( + ) :
   ^a * ^a -> ^a) = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo compiletimeVariableType.fsx
$ mono compiletimeVariableType.exe
3 3.0
```

The example demonstrates the statically resolved variable type syntax, `^<ident>`, as well as the use of *type constraints* using the keyword *when*. Type constraints have a rich syntax, but will not be discussed further in this book.[1] In the example, the type constraint `when ^a : (static member ( + ) : ^a * ^a -> ^a)` is given using the object oriented properties of the type variable `^a`, meaning that the only acceptable type values are those, which have a member function `(+)` taking a tuple and giving a value all of identical type, and which where the type can be inferred at compile time. See Chapter 20 for details on member functions.

- type constraints
- when

The use of `inline` is useful, when generating generic functions and still profiting from static type checking. However, explicit copying of functions is often something better left to the compiler to optimize over. An alternative seems to be using runtime resolved variable types with the `'<ident>` syntax. Unfortunately, this is not possible in case of most operators, since they have been defined in the `FSharp.Core` namespace to be statically resolved variable type. E.g., the "+" operator has type `( + ) : ^T1 -> ^T2 -> ^T3 (requires ^T1 with static member (+) and ^T2 with static member (+))`.

[1] Jon: Should I extend on type constraints? It feels like something better left for a specialize chapter on generic functions.