1 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 used 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 **??**.

· primitive types

1.1 Type Abbreviations

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

· type abbreviation

· type aliasing

Listing 1.1: Syntax for type abbreviation.

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

where the identifier is a new name, and the type-name is an existing type or a compound of existing types. Listing 1.2 shows examples of the defintion of several type abbreviations.

Listing 1.2 typeAbbreviation.fsx:
Defining three 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 used as short abbreviations of longer types, and they add semantic content to the program text, thus making programs shorter and 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 result in programs with fewer errors. Type abbreviations also make maintenance easier. For instance, if we at a later stage decide that positions can only have integer values, then we only need to change the definition of the type abbreviation, not every place a value of type `position` is used.

1.2 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 1.3: Syntax for enumerations.

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

An example of using enumerations is given in Listing 1.4.

Listing 1.4 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 this 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 require 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.

1.3 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 1.5: 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 **??** for a discussion on reference types. Since they are immutable, there is no risk of side-effects. As reference types, they only pass a reference when used as arguments to and returned from a function. 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. Discriminated unions can also be represented as structures using the `[<Struct>]` attribute, in which case they are value types. See Section 1.5 for a discussion on structs.

· The Heap

An example just using the named cases but no further specification of types is given in Listing 1.6.

Listing 1.6 discriminatedUnions.fsx:
A discriminated union of medals. Compare with Listing 1.4.

```
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 1.4, 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 **??** 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 used in discriminated unions can hold any value specified at the time of creation. An example is given in Listing 1.7.

Listing 1.7 discriminatedUnionsOf.fsx:
A discriminated union using explicit subtypes.

```fsharp
1 type vector =
2   Vec2D of float * float
3   | Vec3D of x : float * y : float * z : float
4
5 let v2 = Vec2D (1.0, -1.2)
6 let v3 = Vec3D (x = 1.0, z = -1.2, y = 0.9)
7 printfn "%A and %A" v2 v3
```

```
1 $ fsharpc --nologo discriminatedUnionsOf.fsx
2 $ mono discriminatedUnionsOf.exe
3 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. The names are also called field names. The field names may be used when creating discrimated union values, as shown in Line 6. When used, then the arguments may be given in arbitrary order, nevertheless, values for all fields must be given.

Discriminated unions can be defined recursively. This feature is demonstrated in Listing 1.8.

Listing 1.8 discriminatedUnionTree.fsx:
A discriminated union modelling binary trees.

```fsharp
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 1.1.

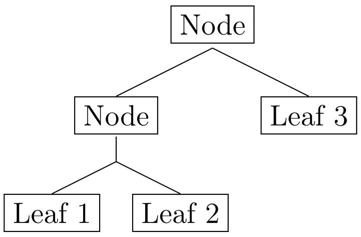

Figure 1.1: The tree with 3 leaves.

Pattern matching must be used in order to define functions on values of a discriminated union. E.g., in Listing 1.9 we define a function that traverses a tree and prints the content of the nodes.

**Listing 1.9 discriminatedUnionPatternMatching.fsx:
A discriminated union modelling binary trees.**

```fsharp
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
```

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

1.4 Records

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

Listing 1.10: Syntax for defining record types.

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

Records are collections of named variables and values of possibly different types. They are reference types, and thus their content is stored on *The Heap*, see **??** 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 1.5 for a discussion on structs. An example of using records is given in Listing 1.11. The values of individual members of a record are obtained using the "`.`" notation

· The Heap
· struct records
· .@.

Listing 1.11 records.fsx:
A record is defined for holding information about a person.

```fsharp
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
```

This examples illustrate a how record type is defined to store varied data about a person, and how a value is created by a record expression defining its field values.

If two record types are defined with the same label set, then the latter dominates the former. This is demonstrated in Listing 1.12.

Listing 1.12 recordsDominance.fsx:
Redefined types dominate old record types, but earlier definitions are still accessible using explicit or implicit specification for bindings.

```fsharp
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 built-in `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 of `RecordsDominance+author` type. The built-in `GetType()` method is inherited from the base class for all types, see **??** 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, i.e., `{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 1.13.

· with@`with`

Listing 1.13 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.

1.5 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, they are best used for small amounts of data. The syntax for defining struct types are:

· structures
· structs

Listing 1.14: 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. Like `let`, the keyword `val` 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` or `do`-bindings are allowed in structure definitions. Fields are accessed using the "." notation. An example is given in Listing 1.15.

· val@`val`
·
· new@`new`
· constructor
· .@.

**Listing 1.15 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 of the inherited *ToString()* function. This is demonstrated in ??.

· overload
· override
· ToString@`ToString(`