Chapter 5

Programming with Types

Abstract In the previous chapter, we took the first step in organizing code such that it better reflects the solutions we seek, is reusable, and is easier to find possible errors in. A fundamental structure in all of this is types, which much like sets in mathematics, form the basic building blocks of what we express in code. In this chapter, we will focus on making new types to better express our solutions. We will examine how to combine types to express higher-order information such as

- defining new types that simultaneously combine existing types.
- define alternatives, i.e., a type that can be one of several other types.

After you have read this chapter, you will be able to model values that contain

- a combination of types such as an address consisting of both street names as a string and zip codes as an integer.
- an alternative list of types such as a shape being either a circle parametrized by its center and radius or square parametrized by its four corners.

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 Chapter 15.

5.1 Type Products: Tuples

Tuples are a direct extension of constants. They are immutable and have neither concatenations nor indexing operations. Tuples are unions of immutable types and have the following syntax:

Listing 5.1: Tuples are a list of expressions separated by commas.

```
<expr>{, <expr>}
```

Tuples are identified by the "`,`" lexeme and often enclosed in parentheses, but that is not required. An example is a triple, also known as a 3-tuple, `(2,true,"hello")`. In interactive mode, the type of tuples is demonstrated in Listing 5.2. The values

Listing 5.2: Tuple types are products of sets.

```

> let tp = (2, true, "hello")
printfn "%A" tp;;
(2, true, "hello")
val tp: int * bool * string = (2, true, "hello")
val it: unit = ()
```

`2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F#, we see that the tuple is inferred to have the type `int * bool * string`. The "`*`" denotes the Cartesian product between sets. Tuples can be products of any type and follow the lexical scope rules like value and function bindings. Notice also that a tuple may be printed as a single entity by the `%A` placeholder. In the example we bound `tp` to the tuple. The opposite is also possible, as demonstrated in Listing 5.3. In this example, a function is defined that takes 1 argument, a 3-tuple. If we wanted a function with 3 arguments, then the function binding should have been `let deconstructNPrint a b c = ...`. The value binding `let (a, b, c) = tp`, binds a tuple with 3 named members to a value, thus deconstructing it in terms of its members. This is called pattern matching. Since we used the `\%A` placeholder in the `printfn` function, the function can be

Listing 5.3: Definition of a tuple.

```
> let deconstructNPrint tp =
    let (a, b, c) = tp
    printfn "tp = (%A, %A, %A)" a b c
deconstructNPrint (2, true, "hello")
deconstructNPrint (3.14, "Pi", 'p');;
tp = (2, true, "hello")
tp = (3.14, "Pi", 'p')
val deconstructNPrint: 'a * 'b * 'c -> unit
val it: unit = ()
```

called with 3-tuples of different types. F# informs us that the tuple type is variable by writing `'a * 'b * 'c`. The "`'`" notation means that the type can be decided at run-time, see Section 5.5 for more on variable types.

Pairs or 2-tuples are so common that F# includes two built-in functions, *`fst`* and *`snd`*, to extract the first and second element of a pair. This is demonstrated in Listing 5.4.

Listing 5.4 pair.fsx:
Deconstruction of pairs with the built-in functions `fst` and `snd`.

```
let pair = ("first", "second")
printfn "fst(pair) = %s, snd(pair) = %s" (fst pair) (snd pair)
```

```
$ fsharpc --nologo pair.fsx && mono pair.exe
fst(pair) = first, snd(pair) = second
```

Tuples of equal lengths can be compared, and the comparison is defined similarly to string comparison. Tuples of equal length are compared element by element. E.g., `(1,2) = (1,3)` is false, while `(1,2) = (1,2)` is true. The "<>" operator is the boolean negation of the "=" operator. For the "<" , "<=", ">", and ">=" operators, the strings are ordered lexicographically, such that `('a', 'b', 'c') < ('a', 'b', 's') && ('a', 'b', 's') < ('c', 'o', 's')` is true, that is, the "<" operator on two tuples is true if and only if the left operand should come before the right when sorting alphabetically. See Listing 5.5 for an example. The algorithm for deciding the boolean value of `(a1, a2) < (b1, b2)` is as follows: we start by examining the first elements, and if `a1` and `b1` are different, then the result of `(a1, a2) < (b1, b2)` is equal to the result of `a1 < b1`. If `a1` and `b1` are equal, then we move on to the next letter and repeat the investigation. The "<=", ">", and ">=" operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable. This is demonstrated in Listing 5.6. However, it is possible to define a mutable variable of type tuple such that new tuple values can be assigned to it, as shown in Listing 5.7. Mutable tuples are value types, meaning that binding to new names makes copies, not aliases, as demonstrated in Listing 5.8. The use of tuples shortens code and highlights semantic

Listing 5.5 tupleCompare.fsx:
Tuples comparison is similar to string comparison.

```
let lessThan (a, b, c) (d, e, f) =
  if a <> d then a < d
  elif b <> e then b < d
  elif c <> f then c < f
  else false

let printTest x y =
  printfn "%A < %A is %b" x y (lessThan x y)

let a = ('a', 'b', 'c');
let b = ('d', 'e', 'f');
let c = ('a', 'b', 'b');
let d = ('a', 'b', 'd');
printTest a b
printTest a c
printTest a d
```

```
$ fsharpc --nologo tupleCompare.fsx && mono tupleCompare.exe
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

Listing 5.6 tupleOfMutables.fsx:
A mutable changes value, but the tuple defined by it does not refer to the new value.

```
let mutable a = 1
let mutable b = 2
let c = (a, b)
printfn "%A, %A, %A" a b c
a <- 3
printfn "%A, %A, %A" a b c
```

```
$ fsharpc --nologo tupleOfMutables.fsx && mono
   tupleOfMutables.exe
1, 2, (1, 2)
3, 2, (1, 2)
```

content at a higher level, e.g., instead of focusing on the elements, tuples focus on their union. While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, where an unprepared reader of the code may not easily understand the computation nor appreciate its elegance without
★ an accompanying explanation. Hence, **always keep an eye out for compact and concise ways to write code, but never at the expense of readability.**

Listing 5.7 mutableTuple.fsx:
A mutable tuple can be assigned a new value.

```
let mutable pair = 1,2
printfn "%A" pair
pair <- (3,4)
printfn "%A" pair
```

```
$ dotnet fsi mutableTuple.fsx
(1, 2)
(3, 4)
```

Listing 5.8 mutableTupleValue.fsx:
A mutable tuple is a value type.

```
let mutable pair = 1,2
let mutable aCopy = pair
pair <- (3,4)
printfn "%A %A" pair aCopy
```

```
$ dotnet fsi mutableTupleValue.fsx
(3, 4) (1, 2)
```

5.2 Type Sums: 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 5.9: 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*. 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.

An example just using the named cases but no further specification of types is given in Listing 5.10. Here, we define a discriminated union as three named cases

Listing 5.10 discriminatedUnions.fsx:
A discriminated union of medals.

```
1 type medal =
2   Gold
3   | Silver
4   | Bronze
5
6 let aMedal = medal.Gold
7 printfn "%A" aMedal
```

```
1 $ fsharpc --nologo discriminatedUnions.fsx && mono
     discriminatedUnions.exe
2 Gold
```

signifying three different types of medals. A commonly used discriminated union is the *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 5.11. In this case, we define a discriminated union of two and three-dimensional vectors.

Listing 5.11 discriminatedUnionsOf.fsx:
A discriminated union using explicit subtypes.

```
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 && mono
     discriminatedUnionsOf.exe
2 Vec2D (1.0, -1.2) and Vec3D (1.0, 0.9, -1.2)
```

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 discriminated 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.

5.3 Records

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

Listing 5.12: 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*. Records can also be *struct records* using the `[<Struct>]` attribute, in which case they are value types. An example of using records is given in Listing 5.13. The values of individual members of a record are obtained using the *"."* notation This example illustrates

Listing 5.13 records.fsx:
A record is defined as 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
```

how record type is used to store varied data about a person.

If two record types are defined with the same label set, then the latter dominates the former. This is demonstrated in Listing 5.14. 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 `person` 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+person` type. The built-in

Listing 5.14 recordsDominance.fsx:
Redefined types dominate old record types, but earlier definitions are still accessible using the explicit or implicit specification for bindings.

```
1  type person = { name : string; age : int; height : float }
2  type teacher = { name : string; age : int; height : float }
3
4  let lecturer = {name = "Jon"; age = 50; height = 1.75}
5  printfn "%A : %A" lecturer (lecturer.GetType())
6  let author : person = {name = "Jon"; age = 50; height = 1.75}
7  printfn "%A : %A" author (author.GetType())
8  let father = {person.name = "Jon"; age = 50; height = 1.75}
9  printfn "%A : %A" author (author.GetType())
```

```
1  $ fsharpc --nologo recordsDominance.fsx && mono
      recordsDominance.exe
2  { name = "Jon"
3    age = 50
4    height = 1.75 } : RecordsDominance+teacher
5  { name = "Jon"
6    age = 50
7    height = 1.75 } : RecordsDominance+person
8  { name = "Jon"
9    age = 50
10   height = 1.75 } : RecordsDominance+person
```

`GetType()` method is inherited from the base class for all types, see Chapter 15 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 the individual members of the source or using the short-hand *`with`* notation. This is demonstrated in Listing 5.15. 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.

Listing 5.15 recordCopy.fsx:
Bindings are references. To copy and not make an alias, explicit copying must be performed.

```
type person = {
  name : string;
  mutable age : int;
}

let author = {name = "Jon"; age = 50}
let authorAlias = author
let authorCopy = {name = author.name; age = author.age}
let authorCopyAlt = {author with name = "Noj"}
author.age <- 51
printfn "author : %A" author
printfn "authorAlias : %A" authorAlias
printfn "authorCopy : %A" authorCopy
printfn "authorCopyAlt : %A" authorCopyAlt
```

```
$ fsharpc --nologo recordCopy.fsx && mono recordCopy.exe
author : { name = "Jon"
  age = 51 }
authorAlias : { name = "Jon"
  age = 51 }
authorCopy : { name = "Jon"
  age = 50 }
authorCopyAlt : { name = "Noj"
  age = 50 }
```

5.4 Type Abbreviations

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

Listing 5.16: 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 5.17 shows examples of the definition of several type abbreviations. 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

Listing 5.17 typeAbbreviation.fsx:
Defining four type abbreviations, three 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
```

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.

5.5 Variable Types

An advanced topic in F# is *variable types*. There are three different versions of variable types in F#: *runtime resolved*, which have 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 5.18. In this example, the

Listing 5.18 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
```

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 5.18 illustrates a very complicated way to add two numbers. The "+" operator works for both types out of the box, so why not something simpler like relying on the F# type inference system by not explicitly specifying types, as attempted in Listing 5.19? Unfortunately, the example fails to compile, since the type

Listing 5.19 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'
```

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 5.20. In the example, adding

Listing 5.20 variableTypeInline.fsx:
The keyword `inline` forces static and independent inference each place the function is used. Compare to the error case in Listing 5.19.

```
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
```

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 5.21. The example

Listing 5.21 compiletimeVariableType.fsx:
Explicitly spelling out of the statically resolved type variables from Listing 5.19.

```
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
```

in Listing 5.21 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. 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 the identical types, and where the type can be inferred at compile time. See Chapter 15 for details on member functions.

The `inline` construction 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. An alternative seems to be using runtime resolved variable types with the `'<ident>` syntax. Unfortunately, this is not possible in the case of most operators, since they have been defined in the `FSharp.Core` namespace to be statically resolved variable types. E.g., the "+" operator has type `( + ) : ^T1 -> ^T2 -> ^T3 (requires ^T1 with static member (+) and ^T2 with static member (+))`.

Discrimate Unions and type abbreviations can be generic as well. For example, in Listing 5.22, we demonstrate how an option-like wrapper can be made for any type. As shown here, the variable type `'a` is first fixed, when a value of the `myOption` is created, and in contrast to function types that are statically resolved, the same definition can be reused for different types of discriminated unions. Similarly, in Listing 5.23, we give an example of a variable type abbreviation. Here, the variable type is a function, which takes a list of some type and returns an value of the same type. For example, `head` returns the first element of any list type, and `avg` returns the average value of lists of floats. For a more in-depth example of generic types, see Section 9.5.

Listing 5.22: A variable discrimated union.

```
> type myOption<'a> = Value of 'a | Error
let intOption = Value 1
let charOption = Value 'a';;
type myOption<'a> =
  | Value of 'a
  | Error
val intOption: myOption<int> = Value 1
val charOption: myOption<char> = Value 'a'
```

Listing 5.23: A variable type abreviation.

```
> type summarize<'a> = 'a list -> 'a
let head : summarize<'a> = List.head
let avg : summarize<float> = List.average
let lst = [0.0..3.0]
printfn "head lst = %A" (head lst)
printfn "avg lst = %A" (avg lst);;
head lst = 0.0
avg lst = 1.5
type summarize<'a> = 'a list -> 'a
val head: summarize<'a>
val avg: summarize<float>
val lst: float list = [0.0; 1.0; 2.0; 3.0]
val it: unit = ()
```

5.6 Key Concepts and Terms in This Chapter

In this chapter you have learned about:

- the **product type** also known as a **tuple**, which is equivalent to a set product;
- the **sum type** also known as a **discriminate union**;
- the **records** which are similar to tuples, but allows you to name the entries
- and as an advanced topic, you have seen F# has flexibility in specifying types either at compile or runtime.