Introduction

In this course we will discuss how to extend our toolbox in order to reduce the number of errors made while programming. We will study a set of tools, known as *compilers*, which are made to analyse our code in order to spot some "obvious" mistakes, before having to run the program.

Programs written using a compiler in between coding and execution are usually protected from a series of bugs, just as if they had been thoroughly tested by a real user. These programs are run twice. The first time they are run is by the compiler, according to an execution model known as *type checker*. This execution model is only a simulation of the actual execution that will later take place. The second time they are run is by the user itself, and this is the actual execution of the program.

As a concrete example we will introduce C#, a programming language built around the premise of static typing. C# also offers another interesting aspect: it integrates together multiple different programming paradigms. One of these paradigms, popularized in the '90s, is object oriented programming, and still enjoys quite some relevance in the current world of software development. The object oriented paradigm will be one of the central focus points of this text.

The choice of C# is mostly motivated by didactic reasons. An obvious contender would be the Java programming language, which offers many of the same features and design principles than C#. Java is also a bit older than C#. The evolution and design of Java, while of unquestioned quality, has been often riddled with compromises in the name of enterprise stability. For this reason, Java has **grown** as much as it has been designed, leading to some very complex choices that make the language very difficult to fully master. Given the didactic nature of the present text, and without taking anything from the quality of Java as a language, we will avoid this complexity.

The interested reader might look into the following topics to better understand the differences between C# and Java from a language perspective:

* custom value types;
* value types as generic arguments;
* sequence generators (yield return);
* inner classes vs lambda's;
* Func vs Function, BiFunction, Predicate, ...;
* properties;
* reified generics (\*this is a very big difference!\*);
* operator overloading.

The list above gives an idea of the aspects of the Java language which are quite complex when compared to C#. Note that we are not in any way implying that knowledge of Java is somehow outdated or useless in the practice: au contraire, countless companies are relying on Java to build excellent and successful products, and the language has a much smaller impact than the quality and passion of the professionals wielding it.

Where we stand

What we have learnt so far are some fundamental aspects of programming with variables and functions. We have seen how we can define and use *variables*, the various *expressions* we can manipulate, and basic *instructions* to *control the flow* of a program based on dynamic conditions depending on our variables. This led us to build some primitive but robust programs, albeit a bit verbose.

We then moved to the definition of "custom instructions", in the form of *functions*. Functions, a seemingly innocuous concept, showed a much deeper potential. Thanks to functions it became possible to define more and more abstract concepts, some of them very detailed, some of them coarser, and to *compose* them together. This way we replicated the innocent simplicity of concepts such "adding two numbers produces a number", but on a much grander scale: "adding two functions produces a function". We achieved even more. Just like there are multiple ways to compose numbers together (`+`, `-`, `\*`, ...), we defined multiple ways of composing functions together (`then`, `repeat`, `AND`, ...). This led us to discover a new space of design of our programs.

Unfortunately, when putting this into practice, it is very much possible to make mistakes. Mistakes are usually found through testing, and can only be prevented with careful discipline. This is not really the ideal way to work: discipline can easily disappear when one is tired, because of lacking communication or mutual understanding within a team, and in general anything depending on human factors will yield fuzzy results.

Course structure

The first part of this text will focus on introducing *type checking*, which is an automated way to prevent many of these typical, small scale errors.

We will then apply our knowledge to a programming language, C#, which is designed around type checking. We will bridge the known constructs studied so far in Python, but within C#. We will also see some new constructs, such as tuples and records, in order to extend our toolbox right away.

C# has a sophisticated mechanism for managing memory. Memory is partitioned in three blocks: the usual space for global variables, the stack for function arguments, and an extra area, the \*heap\*, for the (semi) permanent storage of large pieces of information. We will see some data that goes in the heap, focusing on the example of arrays, and their operators.

In order to better partition applications into reusable modules, C# offers the mechanism of *classes*. Classes are small programs, with their own global variables and functions, defined in isolation from the rest of the application. Such small, isolated programs can then cooperate with each other. From their orchestration emerges a large scale application. Within each class we can, and will, keep using the same constructs that we have seen previously, but limited to the scale (in terms of length of the code) we have seen so far. The samples we have seen until now have, in fact, a size that makes the code readable and understandable. Those samples achieve a sweet spot between richness of the concepts represented and simplicity. We want to maintain this ideal balance and thus strive towards this same amount of code even for much bigger programs. This higher global complexity within the boundaries of equally simple code will come from the use of classes.

Classes may also be extended into new classes, by means of mechanisms such as inheritance. Subtyping polymorphism allows an instance of a class to perform multiple duties, depending on the chain of inheritance. This way, we can define familiar taxonomies in which something also is something else (usually more generic): a cat is also a mammal, and can be used in any context in which a mammal was all that was required.

The interaction between a class and the rest of the program is arbitrated by so called access modifiers. A class, being an isolated unit within the program, can be further segregated from the rest of the application by impeding access to its components from outside. This mechanism ensures that the internals of a class stay neatly hidden, and are not accidentally misused (thereby leading to unexpected behavior and bugs). This mechanism is known as *encapsulation*.

Defining a type checker

We will now study the mechanism of type checking in order to define a coarser, simulated evaluation of our programs to spot potential sources of errors.

Errors known before hand

Consider the following program:

s = input()

n = int(input())

print(s // n)

We cannot know in advance what the inputs will be, but even without knowing what they are, we can immediately see that s will always be a string, for example "Hello world!", whereas n will be a number, such as -3. Trying to divide a string by a number, no matter the string and the number, makes no sense whatsoever. This means that the snippet of code above contains an error, and we do not need to run the program in order to test it: we just know it.

Type checking will try to simulate this process of coarsely tracking the shape of data, in order to detect errors largely in advance of testing the program.

From precise to coarse semantics

Recall that the semantics of a Python program were defined as a function:

eval(P,S)→⟨P′,S′⟩

which would take as input a program P and a state S, and return the new program P′ and the new state S′. The sequence of all states:

S0​→S1​→S2​→…→Sn​

is the execution of the program. Some parts of the state are directly tied to the user input or to the output on the screen, and as such produce effects such as interactive animated behaviors.

Let us now consider a coarser execution model. In this execution model, we replace the state with something similar, but containing less precise information. Instead of storing the value of a variable, such as 0, 1, or one!!!, we will store in the state just its general shape, usually called **type**: `int`, `string`, `bool`, etc.

This way, we will be able to simulate a run of the program, but without being bothered by excessively detailed information. The new state will be called *Typing*, and will usually look like the following:

{x:=int,y:=float,z:=string}

The evaluation function does not really evaluate ("evaluate" comes from Latin, and means "to extract a value from"), but rather checks to make sure that no absurd operations are performed. The eval function will, therefore, be called check, and will be defined as:

check(⟨P⟩,T)→⟨PT​⟩,T′

where P will still be the program, and T will be the current typing, and PT​ is the result of the type checking, namely the type that program P has been determined as having, and T′ will be the new typing.

Type checking

Let us now study some type checking rules.

Variable declaration

Let us begin with variable declaration. In statically typed languages such as C#, variables need to be *declared* before they can be used. The declaration states what type the variable will have throughout its whole lifetime across the whole program. Declaring a variable simply requires stating the type, and then stating the name of the variable, followed by a semicolon. Optionally, the variable may be assigned to an expression right away at the moment of declaration, but this is not necessary. Possible declarations would be:

int x;

int y = 10;

bool z = true;

string s = "Hello statically typed languages!";

The type checking rules of a declaration are simple: they just add to the typings the new variable, as long as it did not already exist in the typings before:

check(⟨X x;⟩,T)→(⟨void⟩,T[x:=X]), provided that x∈/​T. If the variable already existed, then an error is shown to the user and the type checking process is interrupted. Note that void is the type that represents instructions that return no data.

Assigning a variable requires checking whether or not the variable already exists, and if its type is the same (for the moment let us keep it at this, we will extend this concept later) with the type of the expression assigned to it:

check(⟨x = e;⟩,T)→(⟨void⟩,T),

provided that the variable typechecks to a type E:

check(⟨x⟩,T)→(⟨E⟩,T)

and that the expression typechecks to type E as well:

check(⟨e⟩,T)→(⟨E⟩,T).

Type checking of an expression requires type checking its value or its operands. When type checking a value, we can determine its type right away:

check(⟨0⟩,T)→(⟨int⟩,T)

check(⟨1⟩,T)→(⟨int⟩,T)

check(⟨true⟩,T)→(⟨bool⟩,T)

Of course there are infinite such rules (implemented as purely syntactic checks on the individual words of the program), so we do not list them all.

For an example of the type checking of composite expressions, let us consider the type checking of the sum of two values:

check(⟨e1​+e2​⟩,T)→(⟨int⟩,T),

provided that both expressions e1​ and e2​ type-check to the same type int:

check(⟨e1​⟩,T)→(⟨int⟩,T)

and:

check(⟨e2​⟩,T)→(⟨int⟩,T).

The same applies to strings and floating point numbers. For example, for floating point numbers:

check(⟨e1​+e2​⟩,T)→(⟨float⟩,T),

provided that:

check(⟨e1​⟩,T)→(⟨float⟩,T)

and:

check(⟨e2​⟩,T)→(⟨float⟩,T)

The fact that there is no similar rule for boolean values suggests that, should we write something akin to true + false, we would get an error from the type checker.

A similar structure applies to typechecking of boolean conditions. As an example, let us consider the typechecking of the "greater than" (>) condition:

check(⟨e1​>e2​⟩,T)→(⟨bool⟩,T),

provided that check(⟨e1​⟩,T)→(⟨int⟩,T)

and:

check(⟨e2​⟩,T)→(⟨int⟩,T).

The same applies to floating point numbers:

check(⟨e1​>e2​⟩,T)→(⟨bool⟩,T),

provided that:

check(⟨e1​⟩,T)→(⟨float⟩,T)

and:

check(⟨e2​⟩,T)→(⟨float⟩,T)

The fact that there is no similar rule for strings or boolean values suggests that, should we write something akin to true > false or "hello" > "world" (or even 0 > "s"), we would get an error from the type checker.

Some examples

Let us now see some basic examples of type checking:

| **input** | **output** |
| --- | --- |
| ⟨int x;⟩,{} | void,{x:=int} |
| ⟨float y;⟩,{x:=int} | void,{x:=int,y:=float} |
| ⟨3 \* 5⟩,{} | int,{} |
| ⟨y + 5⟩,{y:=int} | int,{y:=int} |
| ⟨x + 5⟩,{x:=string} | error |
| ⟨x := "5"⟩,{x:=int} | error |
| ⟨x := x + 1⟩,{x:=int} | void,{x:=int} |
| ⟨bool b := x > 1⟩,{x:=int} | void,{x:=int,b:=bool} |
| ⟨bool b := x > "hello"⟩,{x:=float} | error |

Getting to know C#

Let us now survey a series of language constructs that we familiarized with in the context of Python, and see how they are translated into C#.

We will begin with various flow and flow control operators.

Conditionals

In Python, we use if-then-else. In C#, just like in Python, we find the same construct. Instead of indentation, C# surrounds statements with curly brackets { and }:

if (Cond) {

THEN-BRANCH

} else {

ELSE-BRANCH

}

The else branch is, of course, optional, so a conditional could also look as follows:

if (Cond) {

THEN-BRANCH

}

Type checking of an if-then-else requires that the condition type checks as a boolean, and the body of both branches both type check as void.

check(⟨if (Cond) {B1​} else {B2​}⟩,T)→⟨void⟩,T,

provided that

check(⟨Cond⟩,T)→⟨bool⟩,T,

and also:

check(⟨B1​⟩,T)→⟨void⟩,T′,

and:

check(⟨B2​⟩,T)→⟨void⟩,T′′.

Note: we will discuss later (in the paragraph about scope) why the typing that results from the if-then-else remains unaltered (T) instead of being some combination of T′,T′′.

Some examples

Consider a couple of illustrative examples.

Suppose we were type checking the following program:

int x = 0;

if (x > 0) {

x = x \* 2;

} else {

x = "";

}

with typing {x:=int}. The type checker would produce an error as result, given that the else branch tries to assign a string to an integer variable.

Even in the case of:

int x = 100;

if (x > 0) {

x = x \* 2;

} else {

x = "";

}

then we would still get an error, since the type checker will take all branches into account, without caring about whether or not the mistaken branch would ever be executed when running the program.

A well-structured conditional which would properly type check would be (starting from a typing of {x:=int}):

int x = 0;

if (x > 0) {

x = x \* 2;

} else {

x = 50;

}

while loops

In Python, we use while-do. In C#, just like in Python, we find the same construct. Just like for if, instead of indentation, C# surrounds statements with curly brackets { and }:

while (Cond) {

BODY

}

Type checking of a while-loop requires that the condition type checks as a boolean and the body type checks to void.

check(⟨while (Cond) {B}⟩,T)→(⟨void⟩,T),

provided that:

check(⟨Cond⟩,T)→⟨bool⟩,T,

and also:

check(⟨B⟩,T)→⟨void⟩,T′.

Some examples

Consider a couple of illustrative examples.

Suppose we were type checking the following program which multiplies all numbers between 1 and 10:

int x = 10;

int n = 1;

while (x > 0) {

n = n \* x;

x = x - 1;

}

with an initial empty typing {} the program above would simply type check correctly, with a final typing of {x:=int,n:=int}.

Modifying the condition to something structurally incorrect, for example comparing x(a number) with "0" (a string), would cause a compiler error:

int x = 10;

int n = 1;

while (x > "0") {

n = n \* x;

x = x - 1;

}

Extra examples

We begin by drawing a rectangle with fixed width and variable height:

string s = "";

int i = 0;

int h = 10;

while (i < 5)

{

s = s + "\*\*\*\*\*\n";

i = i + 1;

}

Very similarly we can draw a rectangle with variable height and width:

string s = "";

int i = 0;

int j = 0;

int h = 5;

int w = 5;

while (i < h)

{

j = 0;

while (j < w)

{

s = s + "\*";

j = j + 1;

}

s = s + "\n";

i = i + 1;

}

Finally, we draw a hollow box with variable width and height:

string s = "";

int i = 0;

int j = 0;

int h = 5;

int w = 5;

while (i < h)

{

j = 0;

while (j < w)

{

if (j == 0 || j == w - 1)

{

s = s + "\*";

j = j + 1;

}

else

{

if (i == 0 || i == h - 1)

{

s = s + "\*";

}

else

{

s =s + " ";

}

j = j + 1;

}

}

s = s + "\n";

i = i + 1;

}

The very last sample is the drawing of a pyramid, according to the familiar algorithm:

string s = "";

int i = 0;

int j = 0;

int w = 9;

bool exit = false;

int stars = 0;

while (!exit)

{

j = 0;

stars = 0;

while (j < w / 2 - i)

{

s = s + " ";

j = j + 1;

}

while (j < w / 2 + i + 1)

{

s = s + "\*";

j = j + 1;

stars = stars + 1;

}

while (j < w)

{

s = s + " ";

j = j + 1;

}

s = s + "\n";

i = i + 1;

exit = stars >= w;

}

for loops

C# also supports other looping constructs, along the tradition of languages such as C and C++. A very well known looping construct is the so-called for loop, which iterates within a given range and also explicitly keeps track of the current position within the iteration. The for loop require three components: an initialization instruction I, a continuation condition Cond, and an increment instruction S, in addition to the body:

for (I; Cond; S) {

B

}

Type checking of a for loop requires that the condition type checks as a boolean, and all other instructions type check to void:

check(⟨for(I;Cond;S){B}⟩,T)→(⟨void⟩,T)

provided that

check(⟨I⟩,T)→(⟨void⟩,T1​)

check(⟨Cond⟩,T)→(⟨bool⟩,T2​)

check(⟨S⟩,T)→(⟨void⟩,T3​)

check(⟨B⟩,T)→(⟨void⟩,T4​)

The type checking rules associated with the body, increment, and condition of a forloop require a little more attention though. A for loop' I instruction is in state of introducing new variables that both the body, the increment and the condition, will all be able to access. Therefore, assume that we have type checked the I instruction, then we type check the body, the condition, and the increment with T1​ instead of T, so that the variables:

check(⟨B⟩,T1​)→(⟨void⟩,TB​)

and:

check(⟨C⟩,T1​)→(⟨bool⟩,TC​)

and also:

check(⟨S⟩,T1​)→(⟨void⟩,TS​)

At a first approximation (we will improve it shortly), the semantics of the for loop are no more than a handy shorthand for the semantics of while:

eval(⟨for(I;Cond;S){B}⟩,T)→eval(⟨I;while(Cond) { B; S }⟩,T).

An example

Consider a while loop such as the one below:

int x = 10;

int n = 1;

while (x > 0) {

n = n \* x;

x = x - 1;

}

There are indeed two instructions which are very closely tied to the body of the loop, to the point that we might want to isolate them syntactically and visually. These instructions are the initialization and increment of x, given that they constitute the fundamental dynamic process that characterizes the loop. This realization suggests that we could translate this while loop in a perfectly equivalent, albeit slightly pleasanter, for loop:

int n = 1;

for(int x = 10; x > 0; x = x - 1) {

n = n \* x;

}

Invalid for loops are easily identified when the equivalent while loop would be invalid as well.

Scope and conditionals

Variables have an own lifetime. The lifetime of a variable determines when the variable becomes available, and when the variable stops being accessible.

The lifetime of variables is usually referred to as **scope**.

Consider for example the following program, assuming a typing of {}:

int x = 10;

if (x > 0) {

int y = 2;

x = x \* y;

} else {

int z = 3;

x = x \* z;

}

It makes intuitive sense that variables y and z will be only available within the respective branches: variable z is neither declared, nor initialized, in the *then* branch, so we cannot use it.

We thus expect that the following piece of code would be rejected by the type check because z is not available in the context of the *then* branch:

if (x > 0) {

int y = 2;

x = x \* y + z;

} else {

int z = -3;

x = x \* z;

}

Variables declared inside the branches will stop being available after the end of the conditional. This means that the following program :

if (x > 0) {

int y = 2;

x = x \* y;

} else {

int z = -3;

x = x \* z;

}

z = y + z;

would also be rejected by the type checker.

This feature is already covered by the typing rules that we have given so far:

check(⟨if(C){BT​} else {BE​}⟩,T)→(⟨void⟩,T)

provided that

check(⟨C⟩,T)→(⟨bool⟩,T)

and also

check(⟨BT​⟩,T)→(⟨void⟩,TBT​​)

check(⟨BE​⟩,T)→(⟨void⟩,TBE​​)

Notice that the whole conditional will return as new bindings the original bindings, thereby restoring them. This means that, even though the branches returned new bindings, respectively TT​ and TE​, these bindings are only active and available within the branch itself, but the local variables defined inside each branch then become unavailable after the conditional.

Semantics of scope

The definition of scope that we have just given for the typechecker does not really match the actual behavior of the semantics as we have seen it so far: adding a variable always wrote either to the globals or to the top of the stack, but now we need a more elaborate behavior. Consider the following snippet:

int x = 10;

if (x > 0) {

string x = "hello";

}

int y = x + 1;

According to our old semantics, we would overwrite variable x with "hello" in the body of the if. When we leave the if, the typechecker is convinced that the variable still has type int, but in the state the int (10) is lost because of the destructive update inside the body of the if.

In order to improve this behavior, we need to turn all variable storage areas in the state into stacks. This means that the global variables, and the stack elements, all become stacks on their own. Entering inside a new scope, for example the then or else blocks of a conditional, pushes a new scope on top of the stack which is the head of the stack (if in a function) or in the globals.

The state for the globals will therefore contain a stack, and thus look (for example inside the conditional above) like:

{h:={x:="hello"},t:={h:={x:=10},t:={}}}

Notice that the state now contains two variables called x, but only the topmost is active. As soon as the conditional is done, then the globals stack is reduced by removing its head, resulting in:

{h:={x:=10},t:={}}

The rules for reading and writing variables will thus look for an appropriate variable as high as possible in this new local stack.

C# is more restrictive than needed in this regard, as two variables with the same name are rejected by the compiler, even though they are in different scopes. There is no real reason to do so beyond aesthetics and a (misguided?) attempt to enforce (best?) practices at the language level. More powerful languages such as F#, Haskell, Scala, and many more allow full usage of variable scope according to our description. We choose thus not to focus on the language, and present the most general understanding of scope currently in broad use.

As a result of this new understanding of how scope affects semantics, the semantic rules for all blocks which might introduce a scope are changed. Notice that wherever we encounter an open curly bracket {, then we must push a new scope. Whenever we encounter a closed curly bracket }, then we must pop the latest scope. We might sum this up as two additional rules:

eval(⟨{⟩,S)→⟨done⟩,S′

where S′ adds a fresh scope to S. When outside a function call (thus S[stack] == {}), then we add a new scope to the globals:

S[globals:={h:={},t:=S[globals]}]→S′

When inside a function call (thus S[stack] != {}), then we add a new scope to the top of the stack:

S[stack:=S[stack][h:=push(S[stack][h])]]→S′

(assuming that push(E) trivially adds an empty scope to the top of the environment).

The } operator does the exact opposite, therefore removing the topmost element from either the globals' stack or the top of the stack.

Scope and loops

The same limitations in scope of variables are of course available in loops. This means that a loop such as:

while (n > 0) {

int k = 2;

x = x \* k;

n = n - 1;

}

given initial typing {x:=int,n:=int} produces as result void and the very same initial typing.

The typing rules responsible for this are:

check(⟨while (Cond) { B }⟩,T)→⟨void⟩,T

provided that:

check(⟨Cond⟩,T)→⟨bool⟩,T

and also:

check(⟨B⟩,T)→⟨void⟩,TB​

Notice that in these rules it is immediately clear how the same bindings T are returned as a result of the type checking of the whole loop, even though type checking the body B of the loop might produce different bindings TB​.

The semantics of for loops require indeed a little bit more attention. Consider a generic for loop: for (I; Cond; S) { B }. We expect that whatever variables declared as part of instruction I will be available to all other instructions (Cond, S, and also B), but after the loop we want those variables to have disappeared. This loop will therefore be translated in a while loop, but encapsulated in an extra scope, given as curly brackets:

{

I;

while (Cond) {

B;

S;

}

}

The extra curly brackets surrounding the generated while loop are therefore responsible for ensuring that, after the loop is finished, the variables introduced by Idisappear from the state.

Additional examples

* Drawing a rectangle with custom width and height. string s=""; int i=0; int h=5; int w=5; while(i<h){ int j = 0; while(j<w){ s = s+"\*"; j = j+1; } s = s+"\n"; i = i+1; }

Drawing a hollow box with variable width and height follows the same structure. Once again, notice that variable j is not present in the state at the end of the program:

string s="";

int i=0;

int h=5;

int w=5;

while(i<h){

int j = 0;

while(j<w){

if(j==0||j==w-1){

s = s+"\*";

j = j+1;

}

else{

if(i==0||i==h-1){

s = s+"\*";

}

else{

s = s+" ";

}

j = j+1;

}

}

s = s+"\n";

i = i+1;

}

Creating new types

The primitive types that the language offers are not the end of the story. If it were so, then it would be very hard to express new concepts such as *a person*, *an order*, *a date*, etc.

C# offers the ability to **define new types by composing existing types**. The simplest way to do so, is by means of tuples.

Tuples

The simplest form of type composition is the *tuple*. Tuples allow us to group together various types into a bigger type which contains all of the grouped types as elements.

A tuple is built from two or more types, say X and Y. By writing (X, Y) we obtain a new type which values will have an instance of X as first component, and an instance of Y as second component. This type can be declared, and its values will look like: `(x,y)`, where `x` has type `X` and `y` has type `Y`. Given a value `v` of tuple type, we can recover its items by writing `v.Item1`, `v.Item2`, etc. This gives rise to some additional typing rules.

The type of a composite expression is a tuple type where the individual items of the tuple have the type given by the corresponding item in the expression:

check(⟨(e1​,…,en​);⟩,T)→(⟨T1​,…,Tn​⟩,T)

provided that

check(⟨ei​⟩,T)→(⟨Ti​⟩,T)

Retrieving a tuple item gives us back an expression which type is the corresponding element in the tuple type:

check(⟨(e.Itemi​);⟩,T)→(⟨Ti​⟩,T)

provided that:

check(⟨e⟩,T)→⟨T1​,…,Tn​⟩,T

Semantics of tuples

Tuples are represented as a binding in the form {1:=v1​,2:=v2​,…}, which will be stored in this very same form in the state.

The semantics of creation of a tuple is therefore the trivial:

eval(⟨(v1​,v2​,…)⟩,S)→⟨{1:=v1​,2:=v2​,…}⟩,S

(assuming that the various arguments have already been evaluated).

The semantics of looking a value up from a tuple are also quite simply translated as binding lookups:

eval(⟨v.Itemi​⟩,S)→⟨V[i]⟩,S

where V is the result of looking up variable v itself:

eval(⟨v⟩,S)→⟨V⟩,S

Example of tuples

Consider now a simple example of tuples used in order to store information about a person:

(string, string) person = ("John", "Doe");

string name = person.Item1;

string surname = person.Item2;

Records and decomposition

It is possible to also give names to the elements of a tuple. Thus, instead of being implicitly called Item1, Item2, etc., the items of the tuple, which is then called a *record*, will take some more domain-appropriate names. We will consider this to be little more than a (very important for readability!) convenience feature, given that it mostly creates aliases for the item accessors.

Another convenience feature, also quite important for the readability of code, is the ability to decompose a given tuple. Suppose we had a tuple e, of type (X, Y, Z), then we could write:

(X x, Y y, Z z) = e;

instead of the much more verbose:

X x = e.Item1;

Y y = e.Item2;

Z z = e.Item3;

Both these features are essentially the same as tuples, with some extra syntactic sugar, so their semantics are left as an exercise to the reader (hint: they are almost the same as those of tuples).

An example

Consider now the case of, for example, defining a person record:

(string name, int age) p = ("Johnny", 93);

Console.WriteLine($"{p.name} has age {p.age}");

Strings prefixed by $ are known as *interpolated strings*. Interpolated strings are built by replacing the expressions between curly brackets with their string representation. The above expression without string interpolation would have to be written by using string concatenation (+) and would therefore look far less readable.

Incorrect variations of this program would, for example, try to access the wrong fields:

(string name, int age) p = ("Johnny", 93);

Console.WriteLine($"{p.surname} has age {p.age}");

Assigning a mismatched tuple would also cause errors:

(string name, int age) p = ("Johnny", "93");

Console.WriteLine($"{p.name} has age {p.age}");

The same program with a tuple, where the items remain nameless, would be:

(string, int) p = ("Johnny", 93);

Console.WriteLine($"{p.Item1} has age {p.Item2}");

Given the significant decrease in readability, there is not much incentive to make use of this version instead of the previous. Tuples will often be found when simply pairing two values together, for example to be returned from a function, whereas records will often be used when the values grouped all have a logical relationship with each other.

As a final note, some versions of C# do not support this syntax for tuples and records, which was added in C# 7.0. For instance, Mono, an open-source branch of .Net Framework, does not support tuples and records in this way. For tuples, you can use the alternative syntax

Tuple<T1,T2,...,Tn> p = Tuple.Create(v1,v2,...,vn);

Example:

Tuple<string,string,int> p = Tuple.Create("John","Doe",34);

Records are not supported, but it is possible to achieve an equivalent data structure by means of Classes or Structures. We will cover these topics further ahead in the course.

Still, we leave it as a matter of taste whether or not tuples or records are used.

An example

In conclusion, the following is an example of C# code that makes use of all the features used. Unfortunately, the code \*needs\* to be surrounded by quite a lot of symbols which are not yet full under our control. This is very unfortunate, given that a professional is expected to understand all he or she is doing, and using something as if it were magic (that is, without understanding it fully) is not what we strive for.

Our example features multiple variable declarations and control flow operators, plus a record to store the current position of the "pixel" being rendered:

int n = 5;

(int, int) size = (n, n);

string s = "";

for (int y = 0; y < size.Item1; y = y + 1)

{

for (int x = 0; x < size.Item2; x = x + 1)

{

if ((x + y \* size.Item2) % 2 == 0)

{

s = s + " ";

}

else

{

s = s + "\*";

}

}

s = s + "\n";

}

Notice that, when compared with Python, both C# and similar languages (such as Java) are much more verbose. The fact that we use curly brackets, and that most coding standards prescribe that curly brackets take a whole line makes our code "grow vertically" quite fast. Moreover, these languages contain many more grammatical elements connected with the type checker. **This verbosity is not inherent to statically typed languages**. Advanced languages with type inference, such as for example Haskell, F#, Scala, or TypeScript, will achieve the same level of safety (and often higher) as C# or Java, but without the same verbosity.

One might even think that verbosity is a feature of these languages.

Dealing with verbosity

Programs written in C# look much more verbose than programs written in Python. This extra verbosity comes from a combination of two factors: the extra curly brackets, which tend to pop up everywhere, and the type declarations next to each variable.

Whereas the curly brackets are no more than a superficial syntactic issue, the type declarations are more annoying: after all, there is no added value to the int part of:

int x = 10;

given that we are assigning an integer (10) right away. This issue is partially taken away by a simple mechanism, called *type inference*, which we can use to let the compiler guess the right type of an expression for us. This is achieved by using var as the type of a variable when declaring and assigning it in one single statement:

var x = 10;

Notice that the type of x is *inferred* to be int, and therefore trying to misuse x as something else will lead us to a compiler error:

var x = 10;

...

x = "hello!"; <---- this gives an error

var is not really a silver bullet, as it does not work on some complex data types such as functions, nor can it be used for function arguments. Still, its presence simplified code quite a lot, by removing some annoying obviousness.