Separating whole sub-programs

So far, we have been working with a single large program where all the data and functions are stored as global elements.

Sometimes a program gets too large and it becomes hard to find the right piece in a large source to modify in order to implement, or modify, a feature. In order to reduce the impact of this issue, a common strategy is to split our program into isolated sub-programs which contain both data and functions. These sub-programs are usually either *modules* or *classes*. Since C# is an object-oriented language, we will be working with **classes**.

A class is defined as a unit containing both data, in the form of a series of variables called **fields**, and actions, in the form of a series of functions called **methods**. Inside methods, we can access the fields of the class or the other methods by invoking them prefixed with the keyword this.

A special method, which has the same name as the class itself, is the **constructor**. The constructor is responsible for initializing the fields of a class. For the next couple of chapters, let us give full access to all our classes by prefixing every method and field with the keyword public.

A simple example

Let us define a simple class which models a person and their initialization:

class Person {

public string name;

public string surname;

public Person() {

this.name = "John";

this.surname = "Doe";

}

}

Creating an instance of a Person uses the new operator, like in the following example:

Person p = new Person();

Of course a person which can only be setup as "John Doe" is not very interesting, so we could give some parameters to the constructor in order to make it possible to specify the name and the surname:

class Person {

public string name;

public string surname;

public Person(string n, string s) {

this.name = n;

this.surname = s;

}

}

Creating an instance now would require specifying the parameters of the constructor just like we specify the parameters of a function:

Person john\_doe = new Person("John", "Doe");

Person jane\_doe = new Person("Jane", "Doe");

Person dr\_strange = new Person("Mister", "Doctor");

We might then add a finishing touch to our class, a method which returns a greeting:

class Person {

public string name;

public string surname;

public Person(string n, string s) {

this.name = n;

this.surname = s;

}

public string Greet() {

return $"{this.name} {this.surname} says hi!";

}

}

After initializing an instance of the class, we might access its fields (given that they are public) and invoke its methods as follows:

Person john\_doe = new Person("John", "Doe");

var name = john\_doe.name;

Console.WriteLine(john\_doe.Greet());

Class semantics

The semantics of classes are quite complicated. There are a lot of moving parts to the behavior of classes, especially in the discovery of the current instance and the management of the special identifier this.

This suggests that classes are not really a beginner's topic, and in order to master them then thought, study, and experience all play an important role.

Note that in the following we will make a strong assumption: all arguments have been evaluated, and so all semantic rules are invoked with values and no sub-expressions, which have already been invoked in an inside-out, left-to-right fashion. This simplifies the definitions, without removing any interesting detail.

Class definition semantics

Defining a class requires storing the class in the state, so that its definition can be found. We will store classes by name in the def portion of the state, according to the following rule:

eval(⟨class C{F1​ f1​,…,C(C1​ c1​,…){CB​},R1​ m1​(P11​ p11​,…){M1​B​},…}⟩,S)→⟨done⟩,S′

In the rule above, we are evaluating the definition of a class named C, with a series of fields f1​,f2​,… respectively with types F1​,F2​,…. The class also features one constructor, obviously called C, with parameters c1​,c2​,… respectively with types C1​,C2​,…, and with a body CB​ made up of a series of instructions. The class also contains a series of methods, called m1​,m2​,…. A method mi​ returns a type Ri​, accepts parameters pi1​,pi2​,…, and has a body Mi​B​.

The new state "simply" stores the class in the definitions in the state, sorting the different elements by category (fields, constructor, and methods):

S[defs:=S[defs][C:={cons:=Cdef​,methods:=Mdef​}]]→S′

provided that the various definitions contain the relevant elements from the class definition:

Cdef​=⟨(c1​,c2​,…)⇒CB​⟩

Mdef​={…,mi​:=⟨(pi1​,pi2​,…)⇒Mi​B​⟩,…}

Notice that we are actively discarding information. We are doing nothing with the fields, for example, given that the fields will be initialised organically by the constructor, and moreover that the class definition does not say anything about their initial value. Further observation of the rules above also shows that we are discarding any type information. Type information is only needed during type-checking, and we can simply assume that if evaluation is actually triggered, then type-checking has succeded and the type information has fulfilled its purpose.

Class construction semantics

Instantiating a class is one of the crucial mechanisms of object oriented languages. It requires making space on the heap, and then using the constructor in order to initialize this space. The constructor accesses the allocated space via this, which is a reference to this new location on the heap. this and the arguments to the constructor are added to the top of the stack. The new location on the heap is known as an **instance** of the class and also known as an **object**.

The call to the constructor is governed by the following semantics which simply return the adjusted memory (where the heap now contains a new instance of the class) and running the body of the constructor:

eval(⟨new C(a1​,a2​,…)⟩,S)→⟨call(CB​;return this;)⟩,SH​

Notice that we are adding a return this statement at the end of the constructor. We want the reference to this to be returned after the constructor is done, but the constructor itself is not allowed to explicitly return anything. This is an example of an *implict behavior*. Object oriented languages are based on multiple such hidden behaviors.

The first step to evaluating construction of a class is to look the class up from the definitions, in order to have access to the body of the constructor:

S[defs][C][cons]→⟨(c1​,c2​,…)⇒CB​⟩

We then create the new instance of the state, by adding a new object to the heap and the parameters to the stack (assume that X is an empty location of the heap).

We begin with the stack:

S[stack:={h:={this:=ref(X),c1​:=a1​,…},t:=S[stack]}]→SS​

and then we fill in the heap:

SS​[heap:=SS​[heap][X:={class:=C}]}]→SH​

Notice that we store the type of the class, C, in the heap. This way, upon looking the object up, it will be possible to know its type and lookup the class definition from the state. The new state SH​ will be used as starting state to evaluate the code of the body of the constructor (CB​), which will initialize the fields of the class with the appropriate values. The semantics rules related to fields (writing them and reading them) are given further below in this section.

Method invocation semantics

Invoking a method is a lot like calling a function, in that the arguments will be added to the top of the stack, bound to the parameter names, and then the result will simply be returned as the body of the method reaches the last instruction.

The major difference comes from the fact that there is an extra parameter which is implicitly added to the stack as well: this. In order to give a value to this, and to know the names of the parameters, we must look in the heap to check the actual type of the class in order to look it up in the definitions.

This is governed by the following semantic rule (where ref(X) is the reference to the heap related to an instance of the class, and m is the name of the method we are invoking):

eval(⟨(ref(X)).m(a1​,a2​,…)⟩,S)→⟨call(Mb​)⟩,S′

Of course we never directly invoke a method from a reference, such as ref(0).m(...). Rather, we will call a method on something that will, eventually, *evaluate to* a reference. For example, we could write v.m(...), where v is a variable that immediately evaluates to ref(0), but we might also write more complex expressions that take a lot of steps to reduce to the reference: `a[0].m(...)`, `f(...).m(...)`, etc. In order to capture all of these cases, we assume that evaluating the left-hand side of the `.` has already happened in previous evaluation steps, resulting in the reference that we can finally use.

Of course the body of the method must be looked up, and to do this we need first to find the class type in the heap thanks to ref(X):

S[heap][X][class]→C

Now that we have the name of the class (C), we can inspect the class definition and find the method body:

S[defs][C][methods][m]→⟨(p1​,p2​,…)→Mb​⟩

At this point we can safely add the arguments and this to the stack:

S[stack:={h:={this:=ref(X),p1​:=a1​,…},t:=S[stack]}]→S′

This way we obtain the final state in which the method body must be evaluated.

Field access semantics

Accessing fields is almost trivial when compared to other aspects of the semantics of classes.

Reading a field requires simply looking up, in the heap, the value of the field at the object pointed at by the reference:

eval(⟨(ref(X)).f⟩,S)→⟨S[heap][X][f]⟩,S

Writing a field requires modifying the heap so that the field of the object pointed at by the reference contains the new value:

eval(⟨(ref(X)).f=e⟩,S)→⟨done⟩,S[heap:=S[heap][X:=Ov​[f:=e]]]

provided that Ov​ is just a shortcut for S[heap][X], that is the object itself.

Class typechecking

The typechecking of classes requires ensuring that the right arguments are passed to methods, and that the methods and fields do indeed exist on the class.

Moreover, the typechecking rules for classes mirror the corresponding semantic rules, just like in general the typechecker is substantially the same as a coarser runtime.

Class definition typechecking

When a class definition is encountered, then its body of the definition is then added to the bindings, "formatted" per field, method, and constructor. The core of the typechecking rule therefore simply changes the typings:

check(⟨class C{ F1​ f1​;… ;R1​ m1​(P11​ p11​, …) {Mb1​​},…,C(C1​ c1​,…) {Cb​} }⟩,T)→⟨void⟩,T′

The new typings will just store an extra symbol, the class itself:

T[C:={fields:={f1​:=F1​,…},methods:={m1​:=Func⟨P11​,…,R1​⟩,…},cons:=Func⟨C1​,…,void⟩}]→T′

Typechecking the bodies

Of course, before being considered completed, the typechecker must also ensure that the different method and constructor bodies are well structured.

This is achieved by invoking the typechecker on the bodies, providing the types of the parameters and this. Since the bodies of the parameters might make use of the definition of the class itself, we will use the typings T′. This way, the definition of the class is "provisionally accepted", even though the bodies of the methods and the constructor are not yet typechecked (and thus the class as a whole has not been confirmed yet).

Each method mi​ is typechecked as follows:

check(⟨Mbi​​⟩,T′[this:=C,pi1​:=Pi1​,…])→⟨Ri​⟩,T′′

The constructor, similarly (besides for the returned type, which is an instance of the class), is typechecked as:

check(⟨Cb​⟩,T′[this:=C,c1​:=C1​,…])→⟨C⟩,T′′′

Construction typechecking

Constructing an instance of a class simply checks whether or not all the given arguments match the type of the corresponding parameter. The parameter types are looked up in the type definitions.

Moreover, invoking the constructor returns a value that has the type of the class itself.

check(⟨new Cn​(a1​,…)⟩,T)→⟨Cn​⟩,T

provided that, for each given parameter ai​:

check(⟨ai​⟩,T)→⟨Ci​⟩,T

assuming of course that the type of the parameters comes from the typings:

T[Cn​][cons]→Func⟨C1​,…,Cn​⟩

Method invocation typechecking

Invoking a method requires ensuring that the arguments passed have the same type as the parameters expected, of course given that the method should exist in the definition.

check(⟨v.m(a1​,…)⟩,T)→⟨R⟩,T

provided that, for each given parameter ai​:

check(⟨ai​⟩,T)→⟨Pi​⟩,T

assuming of course that the type of the parameters and of the result all come from the typings:

T[Cn​][methods][m]→Func⟨P1​,…,R⟩

and that the name of the class, Cn​, comes from the typechecking of v:

check(⟨v⟩,T)→⟨Cn​⟩,T

and subsequently, from the name of the class we can extract the actual definition Cof the class:

T[Cn​]→C

Field access typechecking

Accessing a field simply results in the type of the field as found in the typings:

check(⟨v.f⟩,T)→⟨F⟩,T

provided of course that the type of the field has been looked up from the definition of the class in the typings:

T[Cn​][fields][f]→F

and that typechecking v leads us to the name of the class Cn​:

check(⟨v⟩,T)→⟨Cn​⟩,T

When assigning a field, then we must also check whether or not the expression assigned has the same type as the field:

check(⟨v.f=e⟩,T)→⟨void⟩,T

given that:

check(⟨e⟩,T)→⟨F⟩,T

provided of course that the type of the field has been looked up from the definition of the class in the typings:

T[Cn​][fields][f]→F

and that typechecking v leads us to the name of the class Cn​:

check(⟨v⟩,T)→⟨Cn​⟩,T

A Counter class

Let us now build a very simple example. The class we will define is a counter, which can be incremented by *ticking* it.

class Counter {

int cnt;

public Counter() {

this.cnt = 0;

}

public void tick() {

this.cnt = this.cnt + 1;

}

}

Counter c = new Counter();

c.tick();

c.tick();

Typechecking and running the counter

Typechecking the counter will add the class definition to the bindings, and then typecheck all the methods.

When the typechecker is completed successfully, the code is then ran. Notice how a new Counter object is added to the heap, and all invocations of tick modify the object through the heap itself.

Aliasing

An important aspect of using references, is that we can duplicate the references, which all act on the same object in the heap. Notice how, in the following example, we act on the same object in the heap via two different references:

class MyClass {

int field;

public MyClass(int f) {

this.field = f;

}

public void do\_something() {

this.field = this.field \* 2 + 1;

}

}

var c1 = new MyClass(10);

var c2 = c1;

c1.do\_something();

c2.do\_something();

The Program class and the main method

A special mention goes to the Program class. In languages such as C# and Java, everything is done via objects. This means that the program itself is an object, and it contains a special method called Main. This special method will be automatically invoked by the host runtime:

class Program

{

static void Main(string[] args)

{

...

}

}