Introduction

The essential advantage of having a typechecker is that we can be sure of the quality of our programs. Non existing or wrongly used variables will trigger errors at compile time, so that we are saved from executing a program containing evident mistakes.

This applies very strongly to functions. In the rest of the chapter, we will extend our discussion about scope in order to cover both functions and the lifetime of their parameters as seen by the typechecker.

Static typing and functions

Functions, being custom instructions of sorts, introduce constraints for the type checker. These constraints, given that functions are custom, also take the form of custom constraints. This makes the type-checker somewhat extensible via functions.

A function accepts parameters and returns a value. Each of the parameters, and the returned values, are bound by types, which are declared with a syntax similar to the syntax of variable declarations. In general, a function will contain the following elements:

R f(X x, Y y, ..., Z z) {

...return e...

}

R is the return type, and each returned expression e will need to have type R. Calling the function requires providing arguments, one for each parameter. The types of the arguments must match the types of the parameters given in the function declaration.

The typechecking rules for function declaration simply add the function to the bindings. The type of a function in the bindings requires us to introduce a special syntax. A function that takes as input parameters T1​,…,Tn​, and return type R, has type:

Func<T1, ..., Tn, R>

The body of the function needs to assume type R. This leads us to the following typechecking rule(s):

check(⟨R f(X x, Y y, ...) { B } ⟩,T)→(void,T[f:=Func<X,Y,...,R>])

provided that (at a first approximation at least):

check(⟨B⟩,T[f:=Func<X,Y,...,R>,x:=X,y:=Y,…])→(R,T′)

Notice that we are discarding T′. This is done in order to have no local variables of the function leaving the scope of the function itself. As the type-checker is done with checking the function, then the resulting typings will simply be the original typings before checking the function, plus the function signature itself. Any extra information coming from the typing of the body of the function will not be visible outside and after the declaration.

Calling a function is done just like in languages such as Python, by using the familiar syntax of "function name", "list of arguments between brackets":

y = f(a, b, ...);

Function calls can appear as (parts of) expressions, but also as self-sufficient statements. Type-checking of function calls simply ensures that all arguments have the proper type, and that the resulting type of the expression or statement is the return type of the function:

check(⟨f(a1​,…,an​)⟩,T)→(R,T)

provided that:

check(⟨f⟩,T)→(Func<T1,...,Tn,R>,T)

and, for each argument:

check(⟨ai​⟩,T)→(Ti​,T)

Type checking return

The return statement requires a little extra care when typechecking. Consider the following typechecking rule that we had previously proposed as a placeholder:

check(⟨B⟩,T[x:=X,y:=Y,…])→(⟨R⟩,T′)

The rule above states that the body of a function should typecheck to the type R that the function is supposed to have. Unfortunately, this is not fully accurate, as it would mean that a function such as:

int f() { 1 }

would be acceptable, whereas we only want to accept functions which *return* values of type int, such as:

int f() { return 1; }

This means that we need to distinguish the type of a statement *returning* R, and the type of a statement or expression which *is* an R. We can do this by defining a new type modifier that we "wrap around" R, when returning:

check(⟨return e⟩,T)→(Return(R),T)

provided that the returned expression actually had type R:

check(⟨e⟩,T)→(⟨R⟩,T)

This means that the rule for the typechecking of a function body will be restated as:

check(⟨B⟩,T[x:=X,y:=Y,…])→⟨Return(R)⟩,T′

Control flow and return

Control flow statements, especially if conditionals, must be adjusted to allow returnto happen within the body. For example, consider the following function:

bool even(int x) {

if (x % 2 == 0) {

return true;

} else {

return false;

}

}

This means that, whenever one of the two sides of a conditional returns something of a given type, then the whole conditional will be considered as returning something of that given type.

The type-checking of a conditional will thus be adjusted so as to merge the types returned by the branches:

check(⟨if (C) { P } else { Q } ⟩,T)→⟨R⟩,T

provided that:

check(⟨P⟩,T)→⟨PT​⟩,TP​

and:

check(⟨Q⟩,T)→⟨QT​⟩,TQ​

and also that the types of the branches, when merged, produce a final type which then becomes the type of the whole if conditional:

PT​⊔QT​→R

where ⊔ merges the various cases of branches either producing a Return(T), or void.

If any of the branches returned something, but the other did not, then merging their effects is still considered to return:

Return(T)⊔void→Return(T)

and symmetrically:

void⊔Return(T)→Return(T)

but in the simplest case when neither of the branches returned something, then the whole if is logically considered as to return nothing:

void⊔void→void

and of course when both side :

Return(T)⊔Return(T)→Return(T)

All other cases of ⊔ are considered malformed and will therefore cause a compiler error.

We would end in such a scenario if, for example, we tried to return values of different types in different branches of an if conditional:

if (C) {

return 1;

} else {

return true;

}

The two branches would have type, respectively, Return(int) and Return(bool), which cannot be reconciled into a single type for the conditional which, therefore, cannot pass typechecking.

An exception to the typechecking of the return type rises with functions which return nothing, which are declared as void f(...). Such functions can return an empty expression with statement return;, without arguments. This statement has type Return(void).

Examples of simple functions

Let us begin by defining a function that takes as input three coefficients, a, b, c, a value x, and computes a quadratic formula with them:

int quadratic(int a, int b, int c, int x) {

return a \* x \* x + b \* x + c;

}

int y = quadratic(1, 2, 3, 4);

Using flow control within function also requires no special care. We can see it in action in a function that produces as output a string describing whether or not the input given was even or odd:

string describe(int x)

{

if (x % 2 == 0) {

return "even";

} else {

return "odd";

}

}

var s1 = describe(100);

var s2 = describe(101);

The final example we see is a bit more complex. The factorial function computes the factorial of a given number, using both recursion and control flow:

int factorial(int x) {

var res = 1;

if (x <= 0) {

return res;

} else {

var p = x;

var prev\_x = x-1;

var q = factorial(prev\_x);

res = p \* q;

}

return res;

}

var x = factorial(5);

The body of the function contains a recursive call to factorial itself. Notice that type-checking of the body succeeds because the function itself is added to the typing *before* starting with the type-checking of the bodies. Calls to the function itself will therefore be handled correctly, as long as the function is provided with the required parameters.

As the examples show, there is not much to functions beyond what we have seen so far, and using them is quite similar to what we experiences in previous chapters.

Values of Func type

C#, just like Python, offers the possibility to define functions inline, as special values which we can then just assign to variables and later on invoke.

In C#, these (inline) function values have type Func, and can thus only be assigned to variables of this type. The syntax of such inline functions will usually require a series of parameters (between brackets if there is more than one), then a fat arrow (=>), and finally the expression being returned. For example:

Func<int, int> incr = x => x + 1;

Func<int, int, bool> gt = (x,y) => x > y;

An expression such as (x,y) => x > y is called an anonymous function, a lambda function, and various other names.

Notice that the parameters do not need an explicit type declaration, since the type can be **inferred** from the context.

We can invoke such a function by calling it, for example by writing:

int y = incr(10);

This looks exactly the same as what we would do when invoking a regular function. Indeed, the connection between regular functions and inline functions is very strong (they are the same sort of constructs after all), to the point that regular functions can just be assigned to variables of Func type, as per the following example:

int incr(int x) {

return x + 1;

}

Func<int,int> f = incr;

Anonymous functions typecheck exactly like functions (therefore we will not add any new typechecking rules for them), but with a minor difference: since the arguments do not have type declarations, the types of the arguments come from the type of *context*in which the anonymous function is declared.

Consider for example the following declaration:

Func<int,int> f = x => x + 1;

Variable x, when declared to the left of the function, has no explicit type annotation. From the declaration, it is known that the function will need to have type Func<int,int> (what we just introduced as *the context* of the declaration), thus we know *implicitly* that the function will have an integer argument. Given that x is the only argument of the function, then x must be of type int. Whenever more arguments are present, their types they will be written in the same order. Indeed, consider the example:

Func<int, string, bool, float> f = x,y,z => ...

x is bound to int, y is bound to string, and z is bound to bool.

Examples of lambda functions

As our first, simple example, consider a lambda function that doubles its input number:

Func<int,int> d = x => x \* 2;

Now consider a very similar lambda function that adds 2 to its input number:

Func<int,int> p2 = x => x + 2;

We might now want to redefine the then function that we had seen in Python, this time restricted to integer functions:

Func<int,int> then(Func<int,int> f, Func<int,int> g) {

return x => g(f(x));

}

Notice that the two lambda functions can simply be invoked, just like normal functions.

At this point, we could define, and invoke, the composition of d and p2 via then as follows:

int z = then(d, p2)(5);

We expect z to assume the value 12:

Func<int,int> d = x => x \* 2;

Func<int,int> p2 = x => x + 2;

Func<int,int> then(Func<int,int> f, Func<int,int> g) {

return x => g(f(x));

}

Func<int,int> d\_p2 = then(d, p2);

int z = d\_p2(5);

Currying and closures

When working with lambda functions, new design patterns emerge. Design patterns are best practices that somehow encode a path of least resistance when (a human is) designing code. These paths of least resistance informally suggest that most humans do share a fundamental intuition when writing code, and the resulting broader strategies are usually studied as they represent a very effective cookbook of coding recipes that work in many applied contexts.

One of these patterns is *currying*. Function definitions so far have accepted all the arguments in one single call. Once the call is made, the parameters are bound to the arguments, and the body can be executed. Currying defines functions differently. Their parameters are accepted one at a time, each time invoking the function with yet another parameter.

The signature of such a function, instead of being Func<t, u, v, ...>, which would accept all parameters at once, now becomes Func<t, Func<u, Func<v, ...>...>. Every time we want to pass an argument to the function, then we invoke it, **with that argument only**. Let us consider as an example the definition of a function that adds values together. Thanks to this function, we can derive multiple secondary functions by specifying some, but not all, the parameters:

Func<int, Func<int, Func<int, int>>> add\_mul = x => (y => (z => x \* (y + z)));

Func<int, Func<int, int>> add\_double = add\_mul(2);

Func<int, Func<int, int>> add\_triple = add\_mul(3);

Func<int, int> f = add\_double(4);

int a = f(2);

Func<int, int> g = add\_triple(5);

int b = g(1);

As we invoke function add\_mul for the first time, we only specify the value of parameter x. The result is thus a function (add\_double) which is still a function, but with one parameter less given that now only y and z are left to be specified. Invoking the resulting function again specifies the value of y, yielding yet a new function (f). Invoking this final function produces, at last, the actual value which is the result of the computation.

Notice that, when invoking a curried function, the resulting function that is produced stores the values in the state, in a location called *closure*. The closure are thus all values and variables which the function will later use as all of its parameters will have been specified. After all parameters have been specified, then the closure values are added to the top of the stack for the body of the function to access. We will leave the evaluation rules of function call to the motivated reader.

Notice that management of the closure has no impact on the type-checker, given that the type-checker simply adds all variables it encounters through to the various typings it passes on.

A larger example

In the following, we present a slightly longer example of a function using functions, higher order functions, and recursion in order to define a drawing system:

using System;

namespace Functions

{

class Program

{

static void Main(string[] args)

{

Func<string,string> then(Func<string,string> f,

Func<string,string> g) {

return x => g(f(x));

}

Func<string,string> id = x => x;

Func<string,string> repeat(Func<string,string> f, int n) {

return n <= 0 ? id : then(f, repeat(f,n-1));

}

Func<string,string> symbol(string s) { return x => x + s; }

Func<string,string> star = symbol("\*");

Func<string,string> empty = symbol(" ");

Func<string,string> newline = symbol(System.Environment.NewLine);

Func<string,string> line(int n) { return repeat(star, n); }

Func<string,string> empty\_line(int n) { return repeat(empty, n); }

Func<string,string> square(int n) {

return repeat(then(line(n), newline), n);

}

int size = int.Parse(Console.ReadLine());

Console.WriteLine(square(size)($"The square is:\n\n"));

}

}

}