Storage of large objects

So far, we have only handled small objects. Small objects are simple primitives, which are copied whenever they are passed around.

Consider for example the following program:

var x = 1;

var y = x;

In the program, when assigning x to y, a copy of the value of x is made and stored inside y. Creating a copy of values is handy, given that these values are effectively immutable and cannot change while we do not expect it.

Unfortunately, this mechanism does not scale well. Suppose that x were a sequence of hundreds of thousands of integers. Stating that y = x requires a full copy of the whole sequence and this would cripple the performance of our programs to a halt. For this reason, we introduce the **heap**.

The **heap** is a new location in memory, which will contain data at incremental positions. Variables will be able to store **references** to the heap. An example of state with a heap could be:

{x:=ref(0),y:=ref(1),heap:={0:=1,1:="Hello world!"}}

Suppose that, in this program, we then assigned x to a new variable z (var z = x). Then we would only create a copy of the reference, but not of the actual value, resulting in state:

{x:=ref(0),y:=ref(1),z:=ref(0),heap:={0:=1,1:="Hello world!"}}

Any change to cell 0 of the heap will now be visible through all variables holding a value of ref(0). This mechanism is known as *shared references*.

Great power, great responsibility

Shared references are a source of great power. They make it possible to share complex data, so that changes are immediately visible to all those that hold a reference to the shared data.

This is also one of the greatest sources of very difficult bugs known to programmers of such languages. The ability to hold references to an object can produce scenarios quite distant from our intuitive understanding of reality. Suppose, for example, that both Wim and Pim are holding a reference to a cake, which they safely store in their fridges. Should Wim decide to eat the cake, then the cake in Pim's fridge (which is the same, minus a reference) would also disappear. It is understandable that Pim would then be surprised and, perhaps, frustrated.

Shared references are also a part of our direct experience, but not in the physical world. Suppose that in a family, both husband and wife shared bank cards to the same account. The husband, upon seeing a very pretty cabriolet car with a 3.5L engine, and imagining that this will make his wife very happy, buys the car for a reasonable price of 25,000 euro, thereby almost zeroing the bank account. The wife, unsuspecting, is still convinced that there are 25,000 euro in the bank, but because of the shared reference her expectations are not realistic.

Sharing references is, therefore, an easy way to surprise (usually for the worst) the expectations of some module in our code.

For this reason, it is advised to take great care in understanding what the deeper effect is of sharing references this way.

Arrays

The first big data structure that we will see is the array. An array of type A contains a sequence of elements of type A. The elements of an array can either be retrieved, or stored, by means of an index. The index is a number that tells which element of the array we are referring to:

* 0 for the first element;
* 1 for the second element;
* ...

An array l has l.Length elements. The last element will have index l.Length - 1, given that the array elements are indexed from 0.

An array is created and added to the heap by invoking new A[] { a, b, ..., ... }, where a, b, ... are the elements that must be stored in the array right away. For example, we can create an array containing 5 integer numbers invoking new int[] { 3, 7, 11, 4, 1 }. The length of this array is 5 (as it contains 5 elements). The first element is 3, which has index 0. The second element is 7, which has index 1, and so on. The last element is 1, which has index 4 (length of the array minus one).

The semantic rules governing this process are responsible for:

* evaluating the elements of the array;
* making space on the heap;
* storing the array on the heap;
* returning a reference to the allocated heap space.

Suppose that each element ai​ is a value (otherwise we evaluate them left-to-right, just like function arguments). The semantics of array creation are therefore:

eval(⟨new A[]{a0​,a1​,...}⟩,S)→⟨ref(X)⟩,S′

where X is an unused reference identifier, and S′ has a new element in the heap:

S[heap:=S[heap][X:={0:=a0​,1:=a1​,…}]]→S′

The type rule governing this process is the following:

check(⟨new A[]{a0​,a1​,...}⟩,T)→⟨A[]⟩,T

provided that:

check(⟨ai​⟩,T)→(A,T)

for each element of the array.

Lookup

Arrays are read element-wise, by using the square brackets operator. The syntax is: a[ei​], where a is an expression yielding a reference to an array, and ei​ is an expression yielding an integer which is the position in the array we are supposed to read from.

The semantics of this operator, assuming that both arguments have been evaluated to, respectively, a reference and an integer number, look the array up in the heap and then the right index:

eval(⟨ref(X)[i]⟩,S)→⟨S[heap][X][i]⟩,S

The typechecker is far less exciting. It simply ensures that the first element is an array, whereas the second must be an integer:

check(⟨ea​[ei​]⟩,T)→(A,T)

provided that:

check(⟨ea​⟩,T)→(A[],T)

and:

check(⟨ei​⟩,T)→(int,T)

Storage

Arrays are written element-wise, by using the square brackets operator. The syntax is: ea​[ei​]=ev​, where ea​ yields a reference to an array, ei​ evaluates to an integer which tells us which position in the array to write to, and ev​ yields the value to store into the array.

The semantics of this operator, assuming that both arguments have been evaluated to, respectively, a reference, an integer number, and a value, stores the value in the array in the heap at the right index:

eval(⟨ref(X)[i] = v;⟩,S)→⟨done⟩,S[heap := S[heap][X][i := v]]

The type-checker is, once again, much simpler. It simply ensures that the first element is an array, the second an integer, and that v has the same type as the elements of the array:

check(⟨ea​[ei​]=v⟩,T)→⟨void⟩,T

provided that:

check(⟨ea​⟩,T)→⟨A[]⟩,T

and:

check(⟨ei​⟩,T)→⟨int⟩,T

and also:

check(⟨v⟩,T)→⟨A⟩,T

A simple array

Let us build a very simple array containing the Fibonacci numbers up to a given n. Remember that a Fibonacci number is the sum of the previous two Fibonacci numbers, and that the first two Fibonacci numbers are 0 and 1:

int n = 5;

int[] fibo = new int[n];

fibo[0] = 0;

fibo[1] = 1;

for (int i = 2; i < n; i=i+1)

{

fibo[i] = fibo[i-1] + fibo[i-2];

}

Arrays within arrays

It is also possible to store array references as elements of an array. This allows us to represent multidimensional data structures in a way that faithfully mirrors the multiple dimensions to be modeled.

For example, we could define a screen buffer as a two-dimensional array, where each element of the array becomes a pixel in the screen:

void render(string[][] pixels) {

string buffer = "";

for (int i = 0; i < pixels.Length; i++)

{

for (int j = 0; j < pixels[i].Length; j++)

{

buffer = buffer + pixels[i][j];

}

buffer = buffer + "\n";

}

Console.WriteLine(buffer);

}

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

string[][] screen = new string[size][];

for (int i = 0; i < size; i++)

{

screen[i] = new string[size];

}

for (int i = 0; i < screen.Length; i++)

{

for (int j = 0; j < screen[i].Length; j++)

{

if ((j + i \* screen.Length) % 2 == 0) {

screen[i][j] = "\*";

} else {

screen[i][j] = " ";

}

}

}

render(screen);