Programming Languages: Recitation 12

Goktug Saatcioglu

05.02.2019

1 HW10

Since we have to cover two topics today we will not be going over the solutions to HW10. Please see the solutions available on the classroom Github page. The way to solve all the problems in HW10 are using your understanding of vtables and the mechanism of dynamic dispatch so the solutions should be intuitive.

2 Generics

- In last week's lecture we learned about generics. Let's review the concepts and attempt the generics question in the final exam prep questions.
- Recall that: Generic programming is a style of programming in which algorithms are written in terms of types to-be-specified-later. These type dependencies are expressed using type parameters that are then instantiated when needed for specific types. In other words, generic programming allows you to abstract over types.
- The benefits of generics are
  - Stronger type checks at compile time: the compiler applies strong type checking to generic code and issues errors if the code violates type safety.
  - Elimination of dynamic casts: values can be inserted and extracted from generic data structures without dynamic type checks.
  - Enabling programmers to implement generic algorithms: programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe.
- Generics in Scala (and Java) are implemented using type erasure.
- Autoboxing (coercing primitives to boxed types), unboxing (retrieving the contents of of boxed types) and specialization (telling the compiler not to box).
- Variance:
  - `C[A]` is covariant in `A`: if `S <: T`, then `C[S] <: C[T]`. That is, the subtype relationship between the argument types is preserved by the instantiation of `C`.
  - `C[A]` is contravariant in `A`: if `S <: T`, then `C[T] <: C[S]`. That is, the subtype relationship between the argument types is inverted by the instantiation of `C`.
  - `C[A]` is invariant in `A`: neither `S <: T` nor `C[T] <: C[S]` holds if `S <: T`. That is, there is no subtype relationship between the instantiations regardless of how `S` and `T` are related.
- By default the compiler will treat everything as being invariant.
- Remember that:
  - Each type expression `T` that occurs in a class `C` has an associated variance:
  - If `T` is a parameter type of a method, then this occurrence of `T` is in contravariant position.
  - If `T` is the return type of a method or the type of a val field, then this occurrence of `T` is in covariant position.
  - If `T` is the type of a var field, then this occurrence of T is in invariant position.

  - If `T` is of the form `C2[T1, ..., Ti, ..., TN]` for some generic class `C2` that takes `N` type parameters (e.g. `List[T1]`), then
    * If `C2` is covariant in its ith type parameter, then `Ti` occurs in a position of the same variance as `T`.
    * If `C2` is contravariant in its ith type parameter, then `Ti` occurs in a position of opposite variance as `T` (e.g. if `T` occurs in contravariant position, then `Ti` occurs in covariant position).
    * If `C2` is invariant in its ith type parameter, then `Ti` occurs in invariant position, irregardless of what the variance of the occurrence of `T` is.

- A type parameter `A` of a generic class `C` can be made covariant if it only appears in covariant positions within `C` and similarly, it can be made contravariant if it only appears in contravariant positions. In all other cases, `A` must be invariant.

- Note that the above rules for determining variance of type expressions do not apply to instance private members. That is, if a member is instance private, then all the types in its declaration can be ignored when determining the variance of a type parameter of the surrounding class.

- Intuitively, if `A` occurs in a covariant position, then this expresses a guarantee that the instances of the generic class promise their clients. For example, from a client's perspective, if `A` is the return type of a method m in the generic class then this expresses a lower bound wrt. subtyping on the values that m returns. That is, the client may safely assume that the values returned by calls to m are an instance of any supertype of `A`. From the perspective of the implementation of the class, they express upper bounds with respect to subtyping: the implementation of m must return an instance of some subtype of `A`.

- On the other hand, a contravariant occurrence of a type parameter `A` expresses assumptions that the instances of the generic class make about values provided by the clients. Hence, they express lower bounds with respect to subtyping from the perspective of the implementation of the class. From the clients' perspective, they can be viewed as expressing upper bounds. When a client calls a method m that takes a value of type A as argument, the client is required to provide a value of type `A` or any of its subtypes.

- Lower bounds on type parameters: `B >: A`.

- Upper bounds on type parameters: `B <: A`.

- We can make a contravariant or covariant position no longer occur contravariantly or covariantly by introducing new typer parameters and imposing lower bounds on the introduced parameters. (So as to gaurantee that we won't be just using any unrelated type to our original type parameter.)

- Let's now look at the generics question in the exam prep questions. Consider the Scala code below.

```
class CoVar[+T](x: T) {
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)
}

class ContraVar[-T](x: T) {
  def method1: T = x
  def method2(y: T): List[T] = List(x,y)
}
```

- For each class, indicate the occurrences of the type parameter in the code that violates the variance annotation.

- The occurrence of `T` in `method1` occurs in a covariant position as it is a method return type. The occurrence of `T` in the return of type of `List[T]` occurs in a covariant position as `List` is covariant in its type parameter and `List[T]` is a method return type. The occurrence of `T` in the argument passed into `method2` occurs in a contravariant position as it is the type of the parameter of `method2`.
- Change each class such that the violations of variances are fixed. The only changes you are allowed to do is introduce new type parameters and type bounds and change the parameter/return types of the methods. However, no other change is accepted. If you change any of the types, your implementation must be as precise as possbile so you are not supposed to use `Any` or `Nothing`.
- Let's begin with the `CoVar` class. We need to understand what can go wrong if the current implementation of `CoVar` were to be covariant in its type parameter `T`. Can you give an example of a problematic piece of code that would type check if `CoVar` was to be covariant in its type parameter `T`?
- Consider the following piece of Scala code.

```
var c1: CoVar[String] = new CoVar("hello")
var c2: CoVar[Any] = c  // OK because String <: Any and Covar is covariant
var ls = c2.method2(5)  // OK because c1 is of type Cell[Any] and 5: Int <: Any
ls.last.charAt(0)  // Uh-oh! We called charAt on an Int
```

  Thus, we see that with covariant type parameters subtype relationships due to inheritance can lead to issues.
- How can we fix this code? We introduce a new type parameter `U`. This way we avoid the sub-typing issue we just discovered above. Furthermore, we then need to update the return type `List[T]` to `List[U]` because we now read variables of type `U`. However, for this to work we need to make `U` a supertype of `T` as otherwise we the list we are creating for the return value of `method2` has to have `x` be covariant in type parameter `T`. As a side note, this supertype bound is intuitive as if we make `U >: T` then we avoid the sub-typing issue as described above as the second element of the list, i.e. `y`, will be greater than or equal to `T` so that all of `U`'s methods can be safely used on all variables in the list. The solution code is given below.

```
class CoVar[+T](x: T) {
  def method1: T = x
  def method2[U >: T](y: U): List[U] = List(x,y)
}
```

- Let's move onto the `ContraVar` class. We need to understand what can go wrong if the current implementation of `ContraVar` were to be contravariant in its type parameter `T`. Can you give an example of a problematic piece of code that would type check if `ContraVar` was to be contravariant in its type parameter `T`?
- Consider the following piece of Scala code.

```
var c1: ContraVar[Any] = new ContraVar(new AnyRef())
var c2: ContraVar[String] = c  // OK because String <: Any and ContraVar is contravariant
var s = c2.method1  // OK
s.charAt(0)  // Uh-oh! We called charAt on an AnyRef
```

  Thus, we see that the contravariant type parameter causes a problem.
- Let's try to fix the issue. We introduce a new type parameter `U` where `U` is a supertype of `T`. This way we can return only supertypes of `T` to avoid this issue.

```scala
class ContraVar[-T](x: T) {
  def method1[U >: T]: U = x
  def method2[U >: T](y: T): List[U] = List(x,y)
}
```

However, the compiler will not accept this code. Can you provide an example of why the code is still problematic? Consider the following piece of Scala code.

```scala
var c1: ContraVar[Any] = new ContraVar(new AnyRef())
var c2: ContraVar[String] = c  // OK because String <: Any and ContraVar is contravariant
var s: String = c2.method1[String]  // OK because String >: String
s.charAt(0)  // Uh-oh! We called charAt on an AnyRef
```

So we see that the type bound still can lead to problematic code and we actually need to bound the type `T` at the class declaration level to prevent this issue. The solution code is given below.

```scala
class ContraVar[-T, +U >: T](x: T) {
  def method1: U = x
  def method2(y: T): List[U] = List(x,y)
}
```

Note that we can make `ContraVar` covariant in its type parameter `U`. Can you explain why? Also, notice that this use of contravariance is rather contrived as we can't really use contravariance usefully for this class. However, it is a good exercise to understand generics.

3 Memory Management and Reference Counting

- Recall that there are that the memory used to store the data and code of a program at run-time is split into three parts: the data segment, the stack, and the heap.
- The stack is used to store the values of local variables in function activation records. When it comes to memory management, allocating data on the stack is generally prefered because it has low overhead and the allocated space is automatically freed when the data is no longer needed (e.g. a local variable goes out of scope). However, there are two limitations for stack allocated data: (
  - The lifetime of objects stored on the stack is given by the LIFO order of function invocations. If the lifetime of an object is dynamic and not correlated with any function invocation, then it cannot be allocated on the stack (e.g. function closures).
  - Objects that are allocated on the stack cannot change their size dynamically. For instance, if we want to implement resizable arrays, then such data structures cannot be stored on the stack.
- So if we have objects that must have a lifetime or size that is dynamic must be stored on the heap.
- Broadly, languages fall under two categories. Those that have manual memory management where the heap is managed by the programmer and those that have automatic memory management where the heap is managed by by the language run-time environment.
- Manual memory management is generally more efficient than automatic memory management. However, it is also a source of common bugs such as
  - use after free errors (accessing dangling pointers),
  - double free errors (deallocating objects multiple times),
  - memory leaks (not deallocating objects after they are no longer used).

  These errors can be eliminated or mitigated by relying on automatic memory management techniques. Such techniques include:

- Garbage collection,
- Reference counting,
- Ownership types.

- We showed an implementation of reference counting in C++ using smart pointers. Let's now practice drawing the the memory state of programs that use smart pointers and identifying potential issues that may occur if we do not use smart pointers with care. Recalling our implementation of C++ smart pointers, give the memory state of program given below. Make sure to identify any use of constructors, copy constructors, desctructors and other smart pointer method calls.

```
1   #include "ptr.h"
2   struct A {
3     Ptr<int> x;
4     A(Ptr<int>& _x) : x(_x) { }
5   };
6   int main() {
7     Ptr<int> x = new int (5);
8     Ptr<A> q = new A(x);
9     Ptr<A> p = q;
10    p->x = x;
11
12    return 0;
13  }
```

The memory layout has been given in class. The trace of calls is given below.

1. `Ptr<int>(int*)`: call to `Ptr` constructor when `x` is initialized on line 7.
2. `A(x)`: call to `A` constructor when `A` instance is created on line 8.
3. `Ptr<int>(const Ptr<int>&)`: call to `Ptr` copy constructor when `x` field of `A` is created on line 4.
4. `Ptr<A>(A*)`: call to `Ptr` constructor when `q` is initialized on line 8.
5. `Ptr<A>(const Ptr<A>&)`: call to `Ptr` copy constructor when `p` is initialized on line 9.
6. `Ptr<A>::operator->()`: call to `Ptr` arrow operator when `p->x` is executed on line 10.
7. `Ptr<int>::operator=(const Ptr<int>&)`: call to `Ptr` assignment operator when assignment `p->x = x` is executed on line 10.
8. `~Ptr<A>`: call to `Ptr` destructor when `p` is popped from the stack.
9. `~Ptr<A>`: call to `Ptr` destructor when `q` is popped from the stack.
10. `~A`: call to `~A` destructor from within `~Ptr<A>` destructor of `q` when deleting `*q`.
11. `~Ptr<int>`: call to `Ptr` destructor when `x` is popped from the stack.

- Now that we have a solid understanding of the memory layout of programs using smart pointers and the calls that are being made, let's consider a case where our use of smart pointers can go wrong. Consider the code below.

```
1   #include "ptr.h"
2   int main() {
3     int* x = new int(42);
4     Ptr<int> p = x;
5     Ptr<int> q = x;
6
7     return 0;
8   }
```

This program consistently fails. Can you see why? Identify the error in the program, an error message that we might see on a terminal window (if any) and suggest a fix to the issue.

- Answer: Notice that when we create `p` and `q` we have two different smart pointers pointing to `x`. This means that each one will have a count of 1. When the program exits we first pop `q` off from the stack so the reference count of `q`'s smart pointer will reach 0. So then we deallocate `x` by calling its (implicit) destructor. Now the pointer to `x` from `p` is dangling. So when we pop `p` from the stack the reference count of `p`'s smart pointer will reach 0 and we will call `x`'s destructor. However, we are now dereferencing and deleting a dangling pointer which will thus call delete on an invalid address, which is an operation with undefined behavior. The program may crash or terminate but either way the program has an error. We see that we broke the contract of the smart pointer implementation which requires that the counter value of a smart pointer always reflect the total number of smart pointers pointing to the same heap object. An easy fix is to change line 5 from `Ptr<int> q = x;` to `Ptr<int> q = p;`. (Note that this program is an example of a double free error.)