Case Class Subtyping

Compiler Construction ’13 Final Report

Valentyna Pavliv Jérémie Rossetti

EPFL
valentyna.pavliv@epfl.ch
jeremie.rossetti@epfl.ch

1. Introduction:

During the past semester, we have been building a compiler for Amy language. The compiler is composed by three main parts. Those are the parser, the name analyser and the type checker.
The parser is used to go from text to the code it represents. It uses a tokeniser that changes normal sentences into tokens that can be interpreted afterwards. The name analyser then changes the output of the parser into symbols and populate the symbol table that group all the information provided by the parser.
Then and finally, there is the type checker that, as the name tells, check if the types used are correct and if there are wrong usage of type.

A problem with our current compiler is that we can have classes that extends abstract classes but having two concrete constructable classes is not possible. With our extension, there would be no issue with such behaviour.

2. Examples:

1.
```
object{
   def main() : Unit = {
      operation(new B());
      //Correct, as B is subtype of A
      }

      def operation(op : A) : Unit = {}
}

class A {}

class B extends A {}
```

This example demonstrates that with our extension, we should be able to substitute a class by another one when we are doing calling a method if the class that replace is of a subtype of the parameter type.

2.
```
val y: Some = Some(0)
// Correct, Some is a type
val x: Option = None()
// Correct, because None <: Option
val z: Some = None() // Wrong

y match {
      case Some(i) => () // Correct
      case None() => () // Wrong
}
```

We can see that if we have a class that has two different subtypes, one cannot replace the other. Only the supertype can be replace by both but they should never be able to replace one another.

3. Implementation:

3.1. Theoretical Background:

During the implementation of this part of the project, we used an upper bound check. As we want to know if the type currently doing an operation is a subtype of the one who is expected, we only have to check if it is a type lower in the architecture.

3.2. Implementation Details:

For the implementation, we decided to work mainly on two parts of our compiler because the changes we did were having an enormous impact on a small part of the compiler but used tokens and syntax that already exists. Those parts are the name analyser and the type checker.

First of all, we had a small modification to do inside the symbol table. The way we chose to implement this extension was to create a new map to register the type dependencies and with it the methods that fills it (addTypeDependencies) that add a pair (key, value) of identifiers that represents the relation where the key is the subtype and the value the supertype.

The name analyser is the phase where we populate the symbol table we will use, which contains a map of types, the constructors for the class definition and now the type dependencies map. It is during this part that we changed the name analyser we had. Not only it should add the types into the symbol table but also the type dependencies. The dependency map is then completed whenever we encounter a case class definition during the name analysis process. Instead of only registering the constructor of the class that also stores the parent class, we only need the relation between them. The class that is the parent of the class definition is the supertype that will be the value of a new type that is created with the name of the class.

Subtyping has another massive influence on our compiler: the type checking phase. As the class are not dealt the same way as when there was only the constructor. Now as the classes are considered as types, we can not use the same constraint solver as before. The constraint generation can remain the same as it is only creating a relation where a type has to match another one. However it is not the same solver that has to be used because it is too rigid to the constraints that are generated and should check if the type of the object that was found is a subtype of the type we expect to be there. If it is, then the constraint solver continues, otherwise it throws an error because there is a type mismatch.

4. Possible Extensions:

A possible extension for what we have already implemented would be to have the possibility to be a subtype to more than one class. As an example, we could have a system that could be close to what java have, where we can implement more than one interface. It is quite logical because

sometimes classes is similar or should have a behaviour resembling to more than one other type and thus that would be a great improvement to our compiler.