Polymorphic types

Computer Language Processing '18 Final Report

Sébastien Fulpius Elsa Weber
sebastien.fulpius@epfl.ch elsa.weber@epfl.ch
EPFL

1. Introduction

In the labs we developed a compiler for Amy, which is a simple functional programming language. In Amy, the most interesting language feature is the use of abstract data types (ADT). We define types and then use pattern matching to manipulate them. One of the most commonly used ADT is List.

As Amy is a typed language, ADT are defined for a specific type. For example List is defined for elements of type Int in the Amy standard library. If we want to use a list of String and use all functions defined on List, we have to define all types and functions again. This is very slow and is prone to error as we have to copy some code. To solve this problem, we will introduce polymorphic types. Polymorphic types are types that are defined for one abstract class, case class or function. We define one type or function using one or more polymorphic types. Then we can use this function or class with any types, as long as the usage is coherent with the definition.

2. Examples

Let's get back to our List example. With polymorphic types we can define lists in the following way:

```
abstract class List[T]
case class Cons[T](h: T, t: List[T]) extends List[T]
case class Nil[T]() extends List[T]
```

Here T represents a parametric type, and can represent any existing type in Amy. The capital letter is a convention, and using the same everywhere is clearer, but it works of course for any sequence of characters, so it could have been:

```
case class Nil[manyLettersAndAlso_42]()
  extends List[manyLettersAndAlso_42]
```

What is important is that the same identifier is used for the case class and the abstract class after the extends keyword, even though the abstract class was not defined using that identifier.

There can also be more than one polymorphic type, but then the number must be the same than the parent class. The following is therefore not correct:

```
case class Nil[A, B] extends List[A]
or
case class Nil[A, B] extends List[A, B]
```

Since List has been defined with only one polymorphic type it cannot be extended with two.

Once we defined the classes, they can be used with the following syntax:

```
val x: List[String] = Cons[String]("foo", Nil[String]());
(...)
```

For simplicity, the concrete type **must** be mentioned at every use of a constructor. Additionally, they must be consistent with the one declared in the abstract type, here String, otherwise an error is thrown from the type checker.

Functions using polymorphic types can also be defined. The polymorphic type they use must be declared between brackets just after their name, and it will be the only one recognized for the evaluation of the arguments and the body. To call them, the concrete types which will replace the polymorphic ones must also be specified between brackets, just like for constructors.

```
def head[A](l: List[A]): A = {
  l match {
    case Cons(h, _) => h
    case Nil() =>
      error("Cannot call head on empty list")
  }
}
```

```
head[Int](x)
```

Notice that for the pattern matching, the brackets after the constructors' name are not necessary, they are even not allowed.

One difficulty is that, since polymorphic types can represent any type, even user-defined types, we cannot know *a priori* if what is inside the brackets is an identifier for a polymorphic type, or an ADT. Consider the example:

```
abstract class A

def append(elem: A, l: List[A]): List[A] = {
    Cons[A](elem, l)
}
```

This is a perfectly valid piece of code, where A represents here an ADT and not a polymorphic type. That is why the brackets after the name of the function specifying the polymorphic type used are important, because this is what makes the function polymorphic.

But here if the intention was to use a polymorphic type called A, that is:

```
def append[A](elem: A, l: List[A]): List[A] = {
    Cons[A](elem, l)
}
```

An error would be thrown, because we are not able to differentiate the polymorphic and the concrete type (both called A) anymore.

3. Implementation

3.1 Theoretical Background

Adding type polymorphism is adding the ability for a term to admit "abstract types", or *generic* types as they are sometimes called. This allows to be on a higher level of abstraction, as for some applications the type of the elements that we work on does not matter.

This changes the type system. In function definitions, these types can be manipulated, but without knowing anything about them, so they can only be copied or passed further as arguments (or, possibly compared with equality, in Amy).

One other important change is that different instances of the **same** data structure can now contain elements of different types. With the chosen syntax, it will also help gain in expressiveness, e.g. seeing List[Int] immediately tells us it is a list of integers.

But this affects type checking because knowing that something is of type List, to get back to our main example, doesn't suffice anymore. We need to make a difference between a List of Int and a List of String.

3.2 Implementation Details

3.2.1 Lexing

The only thing needed in the lexing phase is to recognize open and closed bracket ('[', ']') as a token, which was not done before.

3.2.2 Parsing

To allow the use of polymorphic types, we need to make the grammar more flexible for definitions and calls. That is, when defining (or calling) an abstract class, a case class, or a function, allow the use of (possibly many) type parameters between brackets, but also keep the possibility to not write any.

There are two different situations in which we might use polymorphic types. When writing a definition (abstract class, case class or function), only an identifier can be used between the brackets and not real types. On the other hand, when we call these definitions anything can be used, identifiers as well as primitive types. This difference needs to be present in the grammar.

An interesting thing is that, when discovering an identifier between the brackets, it is at this point impossible to determine if it is a polymorphic type identifier or an ADT. That is why we had to create a new type, ClassTypeOrPolymorphic, in addition of the ClassType and PolymorphicType. The final decision can only be made at the Name Analysis phase, when we know more about the environment.

Using the grammar, the parser then builds the AST. We have to somehow convey the information about those types to the next phases of the compiler, so we added a field "polymorphicTypes" to the ClassOrFunDefs. This field is a list of TypeTrees to help keep the positions for error reporting, but they will only contain PolymorphicTypes. We created a function, constructPolymorphicList, to do it.

For case classes we keep the list of polymorphic types of the parent class as well, as it is written after the extends keyword. We do this in order to verify they indeed use the same identifiers, which we will be able to know at name analysis.

We also introduce a list of TypeTrees for the Call, that we simply called ”types” because they can contain any type, polymorphic or not.

3.2.3 Name analysis

This is the most important part for polymorphic types implementation. In this phase, we have to add information about polymorphic types in the symbol table, modify it in the tree nodes of definitions and calls, and check that every aspect of polymorphic types except type checking is correct.

As we add polymorphic types we have to change the function transformType. We pass as argument a map with information about polymorphic types, that will help us going from nominal to symbolic type. When we transform ClassTypeOrPolymorphic, we can now determine if it is a class type or polymorphic type. If we find the type in the symbol table, this is a class type. If not, it should be a polymorphic type and we check that the type is indeed defined using the map given as argument.

When we add definitions in the symbol table, we also add a map from String to Identifier. This allows us to know which polymorphic type Identifier corresponds to a given symbol type. For constructor, we check that the number of polymorphic types is the same as the abstract class type. We also check that the polymorphic types used in constructor definitions are the same as the ones in the extends part of the definition. Finally, once we added all definitions, we check that no polymorphic type has the same name as a definition using what was put in the symbol table.

In the transformDef function, we are getting information from the symbol table and adding it to the symbolic types, so that the same Identifier is used for each type. The interesting part is the transformFunDef function. When we transform the body of the function, we have to know what are the polymorphic types defined in the function. For this we add a third map to transformExpr, where we can look up which types have been declared for this function.

In transformExpr itself, there are only few changes as we already handled most information about types. When we transform a Call, we transform the types used, making them symbolic, to pass them to the type checker.

3.2.4 Type checking

Surprisingly, even though our extension is allowing more expressiveness with types, once in the type checking phase most of the work is already done. What is left to do is to adapt constraints for calls (and the pattern matching), because we need here to check that the definitions are used correctly.

From the information that that we got from the name analyzer, we create a type environment, where each polymorphic type is mapped to the one used for the call (that can also be polymorphic, or not). When generating the constraints, we then call the substitute function, that replaces each type by the one that is equivalent, if present in the type environment. This allows to use different identifiers for the same polymorphic type, for instance calling Cons[B] when Cons was defined as Cons[C].

About the pattern matching, and more precisely changing the CaseClassPattern, since we have more information about the expected type than for calls, we realized types could be inferred and did not need to be specified between brackets. We again call the substitute function for the type constraint, using this time newly created TypeVariables in the type environment.

Because of this, we have to adapt the resolve of the constraints, that is the solveConstraints function. When facing two ClassTypes as ”found” and ”expected” types, we first check they have the same name, and then try to solve constraints on the polymorphic types they might have. By doing so we can determine which are the types used to instantiate the case class in the pattern without needing them to be explicit.

3.2.5 Code generation

Since in Amy all types work the same way, we did not need to change anything in this phase. We could reuse what we did for the labs.

For example, comparison is always done by reference for ADT, which means we can always compare just one value. Pattern matching case classes always has the same structure, independently of the types in the case class. This allows us to use the same code for functions and constructors for any type.

4. Possible Extensions

When we use polymorphic types, we always require to specify the types. This is easier to write a compiler with this restriction, but in general we don’t always need it.

For example, when we instantiate a case class, we write :

```
val x : List[Int] = Nil[Int]()
```

We could easily infer the types : the case class must have the same types as its parent class. We already do this in the pattern matching, and we could also do this in general.

In some case we could even infer the type of the parent class. We could for instance write:

```
val x : List = Cons(2, Nil())
```

Then by looking at the types in the case class, we could know that this is a list of Int. Although this will not work in all cases. If we write:

```
val x : List = Nil()
```

then we don't have enough information to determine the type of the list.